Analyse statique modulaire des langages à objet.

Francesco Logozzo

Thèse

présentée à

l'École Polytechnique

Pour obtenir le titre de

Docteur En Sciences de l'École Polytechnique

Discipline

Informatique

par

Francesco Logozzo

le 15 Juin 2004

Analyse statique modulaire de langages à objets

Modular static analysis of object-oriented languages

Président	Xavier Leroy *Directeur de recherche, INRIA*
Rapporteurs	Agostino Cortesi *Professeur, Università Ca' Foscari di Venezia, Italie*
	Jens Palsberg *Professeur, University of California, Los Angeles, États Units*
Examinateur	David A. Schmidt *Professeur, Kansas State University, États Units*
Directeur de thèse	Radhia Cousot *Directeur de recherche, CNRS*

Acknowledgments

This Ph.D. thesis is the result of the work, the aid and the support of many people, who made it possible to conceive and write it. These people deserve to be acknowledged. I hope I will not forget anyone....

I would like to thank Xavier Leroy, who agreed to be the president of my jury: I am honored of that.

Agostino Cortesi and Jens Palsberg accepted to be the reviewers for my thesis: I am very grateful for the burden they have taken, for their comments and for their suggestions that helped me to improve the present work.

Many thanks also to David Schmidt, for taking part in my jury and for the interesting discussions we had on different subjects, as Italian coffee, Kansas City Chiefs and computer science.

I express all my gratitude to my advisor, Radhia Cousot who did not let me get depressed by my initial failures and supported me throughout all my doctoral activities.

The period that I spent at École Normale Supérieure as a visiting undergraduate student from Scuola Normale Superiore of Pise was very important for my development, as a person and a researcher. I am indebted with Patrick Cousot who patiently answered to all my questions and provided very deep insights in Abstract Interpretation, and to Fabio Beltram who helped me to obtain the grant that made such a period possible.

I would like to thank all the friends who encouraged and helped me: Bruno Blanchet, Charles Hymans, Jérôme Feret, Damien Massé, Laurent Mauborgne, Antoine Miné, David Monniaux, Xavier Rival, Alexander Serebrenik, Axel Simon, Elodie-Jane Sims, Francesco Tapparo, Yann Tholoniat, Eben Upton and Mirko Zanotti.

I would like to thank my parents, Rita and Mimmo, and my two sisters, Erica and Alessandra, for having always supported me. Finally, this thesis (and the work that lead to it) was conceived and written with the constant and tender support and encouragement of Aurélie, for whom I reserve my very special thanks.

Résumé

Dans cette thèse nous présentons un cadre pour l'analyse statique de langages orientés objets qui tient compte des propriétés de modularité de ces langages.

L'analyse statique consiste en la détermination de propriétés d'exécution de programmes. Elle est complètement automatique et couvre toutes les exécutions possibles, à l'opposé du test de programmes. Les propriétés inférées par une analyse statique peuvent être utilisées pour l'optimisation et la vérification. Par exemple, si l'analyse statique d'un programme détermine qu'il ne peut jamais lancer une certaine exception, alors le gestionnaire d'exception correspondant peut être supprimé sans que cela ne présente de danger. De plus, comme la propriété inférées est une approximation sûre du comportement du programme, la spécification du programme peut lui être confrontée afin de vérifier si le programme la respecte.

Une analyse statique est modulaire si le programme à analyser peut être décomposé en composants qui sont analysés séparément et indépendamment et dont les résultats peuvent être fusionnés afin d'obtenir un résultat valable pour le programme entier.

L'étude des langages orientés objet se justifie par le grand impact qu'ils ont eu et qu'ils ont encore sur la technologie informatique. En fait, la programmation orientée objet est fréquemment utilisée dans des domaines allant des cartes à puces (JavaCard) aux grands systèmes distribués (".net Framework" et "Java Enterprise Edition").

Il y a plusieurs défis à relever pour obtenir une analyse statique efficace de langages orientés objet. Tout d'abord, elle doit gérer les particularités de ces langages telles que l'héritage, le polymorphisme et la résolution de méthodes virtuelles. Deuxièmement, elle doit être modulaire. En fait, les programmes orientés objet typiques sont fait de plusieurs milliers de classes et une analyse monolithique du programmes complet peut être trop coûteuse pour être pratiquée. Troisièmement, la technologie orientée objet favorise la programmation par composants, en cela qu'un composant (une classe) est développée une fois pour toute et utilisée dans de nombreux contextes différents. Aussi, une analyse statique efficace doit pouvoir inférée des pro-

priétés des composants valides pour toutes les instantiations possibles de contextes.

Dans cette thèse, nous présentons une analyse qui relève les défis esquissés ci-dessus. En particulier, nous nous concentrons sur une analyse qui peut inférer des invariants de classe. Un invariant de classe est une propriété d'une classe valide pour chaque instanciation, avant et après l'exécution de n'importe quelle méthode de la classe. Notre analyse a plusieurs avantages. Elle est indépendante du langage, elle exploite la structure modulaire des langages orientés objet et elle gère les principales fonctionnalités de ces langages, à savoir l'héritage, le polymorphisme et l'encapsulation

Le cadre présenté dans cette thèse est très flexible. En particulier, il permet de régler finement l'analyse selon les trois axes orthogonaux suivants:

- Domaine abstrait sous-jacent: une classe peut être analysée en utilisant soit un domaine abstrait générique soit un domaine abstrait symbolique de façon à obtenir une analyse plus efficace mais moins précise [74, 76].

- Gestion de l'héritage: une sous-classe peut être analysée soit directement, en expansant syntaxiquement la relation de sous-classe, soit indirectement, en utilisant l'invariant du parent afin d'éviter une explosion quadratique de la complexité [77, 75].

- Traitement des contextes d'instantiation: une classe peut être utilisée soit indépendamment du contexte, afin d'obtenir un résultat valable dans tous les contextes, soit en utilisant une approximation du contexte afin d'obtenir un résultat plus précis mais moins général [78].

Contents

Chapter 1

Introduction

> L'abstraction ne consiste qu'à séparer par la pensée les qualités sensibles des corps, ou les unes des autres, ou du corps même qui leur sert de base.[1]
>
> Denis Diderot (1749)

In this thesis we present a framework for abstract interpretation-based static analyses of object-oriented languages which takes into account the modular features of such languages.

Static analysis consists of determining execution properties of programs. It is completely automatic and it covers all the possible executions, unlike program testing. The properties inferred by a static analysis based on abstract interpretation can be used for optimization and verification. For instance if the static analysis of a program determines that it can never throw a given exception then the corresponding exception handler code can be safely removed. Furthermore, as the inferred property is a sound approximation of the program behavior, the program specification can be matched against it in order to verify whether the program respects or not its specification.

The study of object-oriented languages is justified by the great impact they had, and they are still having, on information technology. For instance a Gartner report [49] has projected that within a few years the two most important object-oriented platforms, .net [82] and J2EE [56], will share the 95% of the distributed systems market. More in general object-oriented

[1] (French) Abstraction consists only in separating the perceptible qualities of bodies with the mind, either one quality from another, or from the bodies to which they apply.

programming is pervasive in different areas, from smart-cards [103] to very large systems.

Modularity is a wished feature for effective static analyses and in particular for the analysis of object-oriented languages. In fact, object-oriented programming favors the development by components: typically, classes are written once and used in many different contexts. For instance, in the .net framework classes are shared by all the programs which support the platform, so that often a class is used in a context very different from the one it was designed for. Furthermore, object-oriented programs may have very complex structures, characterized by a large number of classes and a rather complex interaction web. For instance, NetBeans [87], an open-source Java IDE, is made up of 8328 Java classes with very complex interactions. As a consequence, a class-level modular analysis can be used both to verify components and to reduce the cost of full program analyses.

1.1 Motivations

1.1.1 Object-oriented Languages

Object-oriented languages provide the language support for the object-oriented paradigm. A programming language is said to be object-oriented if it satisfies the three properties of *encapsulation*, *inheritance* and *polymorphism*. Roughly speaking, encapsulations facilitates code modularization, inheritance make possible code sharing and polymorphism enables to call a variety of functions using exactly the same interface. More specifically,

- encapsulation means that the state of an object is not directly accessible by the context. This implies that the only way to access to object data is through messages passing;
- inheritance allows to create objects by specializing or redefining existing ones. The inherited object may answer to all the messages of the parent;
- polymorphism means that the same operation may behave differently on objects of different classes.

There are two main families of object-oriented languages.

The first family contains object-oriented extensions of procedural or functional languages such as C++ [101], Objective-C [5], Object Pascal [12] or OCAML [70]. The advantage of such languages is that they are backward-compatible with existing code, so that they favor a smoothly migration to object-oriented. The drawback is that as they are half-path languages they present some idiosyncrasies, e.g. rather complicated syntax and semantics.

The second family contains *pure* object-oriented languages as Smalltalk [55], Java [56] or C# [84]. The advantage of such languages is that they strongly enroll the object-oriented concepts, so that for instance they ensure encapsulation. The drawback is that they oblige the programmer to work following the object-oriented methodology.

In mainstream languages of both families, objects are intensionally described by classes. A class is a description of the structure and the behavior of set of objects. Classes may specify access permissions for clients and inherited classes, visibility and member lookup resolution. Objects are created through class *instantiation*.

1.1.2 Verification and Optimization

Verification of object-oriented programs is a hard task made harder by some peculiarities of such languages as inheritance, virtual methods and component-based development. Traditional program testing has two drawbacks. First, it is not sound as just a finite number of executions can be covered. Second, the testing of a class is made even harder by the fact that the instantiation context is not known *a priori*. Theorem provers require heavy human interactions, and even for rather simple cases they may entail a considerable human effort, e.g. as in [60]. As a consequence they are not suited for the analysis of large programs. Model checkers suffer of the state-explosion problem, the incapability of checking directly the source code and the limitations to adapt to modules without human-provided annotations [65].

Optimization is a sensible issue in the implementation of object-oriented languages. Dynamic resolutions of method-calls [42], implicit box and unboxing of objects [89], checks of array bounds [3], etc. may cause non-negligible slowdown. Nevertheless, most of the existing optimizations are whole-program, so that e.g. they cannot be used for the optimization of a single class, aside from its context.

1.1.3 Modularity

Modular reasoning is a widespread methodology that has been proved effective in many human activities. It enables to handle complex problems by studying smaller, easier to deal with sub-problems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them.

In computer science it is a main tool for reasoning about parts of the systems, develop components independently, and even configure customized systems. For instance, it has been worthwhile for system designs, as a methodol-

ogy to specify and to document the artifacts of a system under development [97]; for language specification as constructs capable to express and to check some aspects of the modular structure [69]; for compilation as a way to structure the source code and to speed-up the compiling process [17].

A methodology is worthy of being called modular if it satisfies at least the three requirements of *decomposability*, *composability* and *understandability*. We say that a methodology satisfies

- the decomposability requirement if it allows to decompose a complex task into easier sub-problems, connected by a simple structure and mutually independent;
- the composability requirement if it favors the production of elements which may be freely combined with each other to produce new systems;
- the understandability requirement if the elements can be understood with a limited knowledge of the others.

The object-oriented paradigm satisfies the requirements above. In such a view a complex system is decomposed into a set of objects which interact via message passing. Objects are specified and developed independently, so that it is possible to reuse them in contexts different from the one they were originally designed for. Finally, objects have a clear interface, i.e. the messages they can answer, so that they can be understood without a full knowledge of the surrounding context.

1.2 Abstract Interpretation

We use the theoretical framework of Abstract Interpretation. Abstract Interpretation is a general theory due to P. Cousot's *thèse ès sciences mathématiques* [25] which formalizes the notion of *approximation*. In particular, it formalizes the idea that the semantics of a language can be more or less precise depending on the considered observation level. Semantics can be partially ordered according to their relative precision [29, 28] and the more precise the semantics the more precise the properties about programs it captures.

An abstract interpretation-based static analysis is an abstract semantics which is precise enough to capture properties of interest and coarse enough to be computable. Differently stated, a static analysis is a sound, finite and approximate calculation of the program semantics. Those three requirements imply that

- no program behavior is ignored (soundness);

– the static analysis terminates and the inferred property is computer-representable (finiteness);

– the inferred property may not be the most precise one (approximation).

A static analysis is modular if a program can be decomposed into components (decomposability) which are analyzed separately (understandability) and whose results can be merged together in order to obtain a result valid for the whole program (composability).

1.3 Results

We present a modular static analysis for object-oriented languages. In particular we concentrate on the analysis of the basic *bricks* of object-oriented programming, i.e. classes.

1.3.1 Static Analysis of Classes

We introduce a framework for the modular inference of class invariants. A class invariant is a property that it is valid for all the instances of the class, before and after the execution of any method. It can be used for both verification and optimization of object-oriented programs. In fact, as an abstract interpretation-based static analysis is sound, the inferred property is valid for all the object instances and for all the contexts; as static analysis is automatic, classes can be verified without human intervention. Class invariants can be used for the verification of a single class or of a whole program. In the first case, given a class specification, e.g. the absence of run-time errors, it is sufficient to match the class specification against the class invariant in order to see if the class respects or not its specification. In the latter, the cost of the analysis of a whole program can be drastically reduced by performing the analysis of its classes in parallel. Furthermore, the analysis of a class can be used many times. For instance, when analyzing programs that use libraries, class invariants may replace the analysis of the class source code. This reduces the overall cost of the analysis. Finally, class invariants can be used for the optimization of code at the class level. For instance, if a class invariant states that the class will never throw a given exception, then in the corresponding code for throwing/handling the exception can be dropped. Once again, this is a consequence of the soundness of the analysis.

```
class StackError extends Exception {}

class Stack {
  // Inv : 1 <= size &&
  //      0 <= pos <= size &&
  //      size=stack.length

  protected int size, pos;
  protected Object[ ] stack;

  Stack(int size) {
    this.size = Math.max(size,1); this.pos = 0;
    this.stack = new Object[this.size];
   }

  boolean isEmpty() { return (pos <= 0); }
  boolean isFull() { return (pos >= size); }

  Object top() throws StackError {
    if (!isEmpty())
        // 0 <= pos-1 < stack.length
      return stack[pos-1];
    else throw new StackError();
   }
   void push(Object o) throws StackError {
    if(!isFull()) {
        // 0 <= pos < size
      stack[pos++] = o;
    } else throw new StackError();
   }
   void pop() throws StackError {
    if(!isEmpty())
      pos--;
    else throw new StackError();
   }
}
```

Figure 1.1: A class implementing a stack

```
class StackWithUndo extends Stack {

  // SubInv : Inv && -1 <= undoType <= 1 &&
  //       if undoType == 1 then 0 < pos
  //   else if undoType == 0 then 0 <= pos <= size
  //   else if undoType == -1 then pos < size

  protected Object undoObject;
  protected int undoType;

  StackWithUndo(int x) {
    super(x);
    undoType = 0;
    undoObject = null;
  }
  void push(Object o) throws StackError {
    undoType = 1;
    super.push(o);
  }
  void pop() throws StackError {
    if(!isEmpty()) {
       undoType = -1;
       undoObject = stack[pos-1];
     }
    super.pop();
  }
     // StackError never thrown
  void undo() throws StackError {
    if(undoType == -1) {
      super.push(undoObject);
      undoType = 0;
    } else if (undoType == 1) {
      super.pop();
      undoType = 0;
    }
  }
}
```

Figure 1.2: A subclass of `Stack` with undo

1.3.2 Introductory Example

Let us consider the classes in Figure 1.1 and Figure 1.2. The first class, `Stack`, is the implementation of a stack parameterized by its size, specified at object creation time. It provides methods for pushing and popping elements as well as testing if the stack is empty or full. Moreover, as the internal representation of the stack is hidden a stack object can only be manipulated through its methods. The second class, `StackWithUndo` extends the first one by adding to the stack the capability of performing the *undo* of the last operation.

The comments in the figures are automatically derived when our framework is instantiated with the Octagon abstract domain [86] refined with trace partitioning [59]. In particular, for the `Stack` we have been able to discover the class invariant `Inv` without any hypothesis on the class instantiation context. The invariant `Inv` guarantees that the array `stack` is never accessed out of its boundaries. This implies that the out-of-bounds exception is never thrown (verification) so that the bounds checks at lines 22 and 28 can be omitted from the generated bytecode (optimization).

The class invariant for `StackWithUndo`, `SubInv`, states that the parent class invariant is still valid and moreover the field `undoType` cannot assume values outside the interval $[-1, 1]$. It implies that the method `undo` will never raise the exception `StackErr`. Once again this information can be used for verification (if a class never raises an exception `Exc`, then the exceptional behavior described by `Exc` is never shown) and for optimization (as the exception handling can be dropped). It is worth noting that `SubInv` has been obtained without accessing to the parent code but just to its class invariant.

Finally, let us consider the class `Test` in Figure 1.3 which creates an instance of `StackWithUndo`. Using the properties inferred about `StackWithUndo`, and not its code, we can show that the exception `StackError` is never thrown. Therefore, the code inside the exception handler is never reached so that the program always prints "`ok`!" on the console.

1.3.3 Main Results

We present a language-independent static analysis for the analysis of classes. We systematically derive the equations that characterize a class invariant from a concrete and liberal class semantics. As a consequence, the soundness of the approach is guaranteed by construction.

We consider the different aspects/features of object-oriented languages. In fact, the analysis:

– exploits encapsulation for the inference of class invariants;

```
class Test {

   public static void main(String args[]) {

      StackWithUndo s = new StackWithUndo(10);

      try{
         for(int i=0; i < 10; i++)
            if(Math.random()<0.5)
               s.push(new Integer(i));
            else
               s.undo();
      }
      catch(StackError e) {
         // Unreachable code
         System.out.println("Error in stack operations!");
         return;
      }
      System.out.println("ok!");
   }
}
```

Figure 1.3: A class that uses `StackWithUndo`

– supports inheritance as it infers subclass invariants without accessing to the code of the ancestors;

– handles polymorphism by considering a semantic characterization of subtyping.

It is worth noting that the analysis is modular in that it is performed on classes (decomposability), classes are analyzed without any hypothesis on the instantiation context (understandability) and the results of the analysis can be easily combined (composability).

In addition to the inference of class invariants, we present

– a program-transformation for the derivation of *approximate* classes to be used either for documentation or as tester for the class clients;

– a separate compositional analysis for the parallelization of the static analysis of an object and its context.

1.4 Overview of the Thesis

This thesis is made up of 11 chapters. Each chapter begins with an introductory paragraph and a section in which the subject of the chapter is informally introduced and compared with the existing literature. The results of chapters 5 and 8, 6, 7, 9, 10 have been published in the proceedings of international conferences [77, 76, 74, 75, 78].

Below we sketch the content of the next chapters.

In Chapter 2 we recall some basic definitions, notations and results used throughout the thesis.

In Chapter 3 we introduce the concrete class semantics. The semantics of a class is given by the semantics of all its instances. In its turn, the semantics of an object is a set of traces. Each trace corresponds to a possible evolution history of the object internal state. We show the soundness and the completeness of such a semantics w.r.t. a trace semantics for a full object-oriented program.

In Chapter 4 we obtain the reachable-states semantics of a class by stepwise abstraction of the concrete semantics. First, all the traces corresponding to the same interaction with the context are merged together. Then, the states are collected and the reachable-states semantics is obtained by construction.

In Chapter 5 we show how an approximation of the reachable-states semantics is a class invariant and how it can be obtained as soon as a static analysis for the methods bodies is provided. Furthermore, we show how to handle the situation in which an object exposes a part of its state to the context. We illustrate the results by studying in the details the analysis of two classic examples, a bank account and a random walk.

In Chapter 6 we introduce an axiomatic characterization for symbolic relational modular analyses. We prove it correct with respect to a trace semantics, and we show how it can be used to perform an incremental modular analysis of the object methods. It turns out that such an axiomatization captures the structure of a whole class of modular static analyses, namely the symbolic modular ones.

In Chapter 7 we study the situation in which a symbolic relational analysis is used for the approximation of the methods semantics. We show how to derive approximated classes, obtained by replacing the methods with the relations that approximate their input/output behavior. An approximated class can be used for documentation, as a tester for clients that use the class and for deriving two kinds of class invariants.

In Chapter 8 we extend our framework in order to cope with inheritance. Such an extension is worthwhile for the effective analysis of large class hi-

erarchies. We show how to infer an invariant for a subclass using just the parent's invariant and not to its code. We consider the two orthogonal aspects of inheritance: class extension and method redefinition and we illustrate the results on an extension of a stack.

In Chapter 9 we present an application of class invariants to a more semantic characterization of the inheritance relation. Inheritance is defined in terms of preservation of properties: a class is a *semantic* subclass of another if its class invariant preserve the one of the ancestor. Such an approach is effective for the modular analysis of polymorphic functions.

In Chapter 10 we use regular expressions for approximating the interactions between a context and an object. The intuition is that a node in the expression corresponds to an invocation of a set of methods. We show how a generic static analysis of a context that uses an object can be split into two separate semantic functions involving respectively only the context and the object. We introduce an iterative schema for composing the two semantic functions. A first advantage is that the analysis can be parallelized, with a consequent gain in memory and time. Furthermore, the iteration process returns at each step an upper approximation of the concrete semantics, so that the iterations can be stopped as soon as the desired degree of precision is reached.

Finally, in Chapter 11 we report the conclusions and the perspectives for future work.

Chapter 2

Preliminaries

> Resulta que scriptura in symbolos es circa dicem vice plus breve que scriptura per lingua commune. [1]
>
> Giuseppe Peano
> *Formulario Mathematico* (1896)

In this chapter we introduce the mathematical background used in the rest of the thesis. We fix the notation and we recall some well-known results in lattice theory, fixpoint theory and abstract interpretation theory.

2.1 Notation and Basic Definitions

We use the notation of denoting sets with capital letters and elements of sets with the sans serif font. So, when e is a member of the set S we write $\mathsf{e} \in S$ or $S \ni \mathsf{e}$. We denote with $\mathbb{N}$ the set of natural numbers, with $[a..b]$ the set $\{i \in \mathbb{N} \mid a \leq i \leq b\}$, with $\mathbb{Z}$ the set of integer numbers, with $\mathbb{B}$ the set of boolean values and with $\mathbb{O}$ the proper class of all the ordinals [15].

Given two sets U and V, the cartesian product is $U \times V$. The i-th element of a vector $\mathsf{v} = \langle \mathsf{u}_1, \mathsf{u}_2, \ldots \mathsf{u}_n \rangle \in U_1 \times U_2 \cdots \times U_n$ is denoted by $\mathsf{v}_{(i)}$. A relation R *between* U and V is a subset of the cartesian product, $R \subseteq U \times V$, and a relation R *on* U is $R \subseteq U \times U$. We write $\mathsf{u} R \mathsf{v}$ to mean $\langle \mathsf{u}, \mathsf{v} \rangle \in R$. We say that u is in relation R with v if $\mathsf{u} R \mathsf{v}$ holds. Given two relations $R_1 \subseteq U_1 \times V_1$

[1] (Latino sine flexione) It turns out that writing using formulas is almost ten times shorter than using natural languages.

and $R_2 \subseteq U_2 \times V_2$ the relation $R_1 \times R_2 \subseteq (U_1 \times V_1) \times (U_2 \times V_2)$ is

$$\{\langle\langle \mathsf{u}_1, \mathsf{v}_1\rangle, \langle \mathsf{u}_2, \mathsf{v}_2\rangle\rangle \mid \mathsf{u}_1 R_1 \mathsf{v}_1 \wedge \mathsf{u}_2 R_2 \mathsf{v}_2\}.$$

The set of all u which are in relation R with some v is called the domain of R, $\mathrm{dom}(R)$, and it defined as:

$$\mathrm{dom}(R) = \{\mathsf{u} \in U \mid \exists \mathsf{v} \in V.\ \mathsf{u} R \mathsf{v}\}.$$

In a similar way, the co-domain or the range of a relation, $\mathrm{range}(R)$, is defined as:

$$\mathrm{range}(R) = \{\mathsf{v} \in V \mid \exists \mathsf{u} \in U.\ \mathsf{u} R \mathsf{v}\}.$$

An *equivalence* relation E on a set S is a relation on S which is reflexive, symmetric and transitive. Given an equivalence E on S, an equivalence class is $[\mathsf{s}_0] = \{\mathsf{s} \in S \mid \mathsf{s} E \mathsf{s}_0\}$ and the *quotient* set of S by E is defined as:

$$S_{/E} = \{[\mathsf{s}_0] \mid \mathsf{s}_0 \in S\}.$$

2.1.1 Partial Orders

A *partial order* on a set D is a relation $\sqsubseteq$ on D which is reflexive, antisymmetric and transitive. A set with a partial order defined on it is called a partially ordered set, poset, or, with an abuse of language, simply a partial order. We denote it as $\langle D, \sqsubseteq\rangle$. A relation on D which is reflexive and transitive is called a preorder.

Given a poset $\langle D, \sqsubseteq\rangle$ and a subset U of D, we say that an element $\mathsf{d} \in D$ is an *upper bound* for U if $\forall \mathsf{u} \in U.\ \mathsf{u} \sqsubseteq \mathsf{d}$. We say that $\mathsf{d} \in D$ is the *least* upper bound of U, denoted by $\sqcup U$, if it is an upper bound and any upper bound $\mathsf{d}' \in D$ is such that $\mathsf{d} \sqsubseteq \mathsf{d}'$. An element $\mathsf{u} \in U$ is the *largest* element of U if u is an upper bound for U. It is worth noting that because of the antisymmetric property, if the largest element exists then it is unique. If it exists, we denote the largest element of D with $\top$. Lower bounds, greatest lower bounds and smallest elements are defined dually. In particular, if it exists, we denote the smallest element of D with $\bot$.

A poset $\langle D, \sqsubseteq\rangle$ is called a *lattice* if any two elements of D have both a greatest lower bound and a least upper bound. If a poset admits greatest lower bounds and least upper bounds even for infinite sets then it is a *complete* lattice. In such a case we write either $\langle D, \sqsubseteq, \bot, \top, \sqcup, \sqcap\rangle$ or, when the lattice operators are clear from the context, simply $\langle D, \sqsubseteq\rangle$. We say that the operator $\sqcup$ is complete if any subset U of D admits the least upper bound $\sqcup U$.

A poset $\langle D, \sqsubseteq\rangle$ is said to satisfy the ascending chain condition (ACC) [88] if every ascending chain $\mathsf{d}_1 \sqsubseteq \mathsf{d}_2 \sqsubseteq \ldots$ of elements of D is eventually

stationary, that is, there is some positive integer n such that $\forall m > n.\ \mathsf{d}_m = \mathsf{d}_n$. Similarly, $\langle D, \sqsubseteq \rangle$ is said to satisfy the descending chain condition (DCC) if there is no infinite descending chain. Every finite poset satisfies both ACC and DCC.

2.1.2 Functions and Fixpoints

Given two sets D and R, a relation $f \subseteq D \times R$ is called a function, or a mapping, if $\mathsf{d} f \mathsf{r}_1$ and $\mathsf{d} f \mathsf{r}_2$ imply $\mathsf{r}_1 = \mathsf{r}_2$ for any d, r_1, and r_2. In other words, a relation f is a function if and only if for every d from dom(f) there is exactly one r such that $\mathsf{d} f \mathsf{r}$. This unique r is called the value of f at d and it is denoted $f(\mathsf{d})$. We specify functions using the λ-notation [19], i.e. $\lambda x.B$ defines a function with an input x and an output given by the expression B, parametric w.r.t. x. Given a function f, a $\mathsf{y} \in \mathrm{dom}(f)$ and a $\mathsf{v} \in \mathrm{range}(f)$, the update of f is the function defined as:

$$f[\mathsf{y} \mapsto \mathsf{v}] = \lambda \mathsf{x}.\ \text{if } \mathsf{x} = \mathsf{y} \text{ then } \mathsf{v} \text{ else } f(\mathsf{x}).$$

We denote the set of all the functions with a domain D and a range included in R as

$$[D \to R] = \{f \mid f \text{ is a function} \wedge \mathrm{dom}(f) = D \wedge \mathrm{range}(f) \subseteq R\}.$$

Given two functions $f \in [D \to R]$ and $g \in [R \to C]$ their composition is $g \circ f \in [D \to C]$. In particular if the domain and the co-domain of a function f coincide then it can be composed with itself so to obtain: $f^i = \underbrace{f \circ f \cdots \circ f}_{i \text{ times}}$.

The notion of quotient set can be extended to cope with functions. So, let S, R be sets and let $f \in [S \to R]$ be a function. Then the quotient set of S by f is $S_{/E_f}$, where the equivalence relation is defined as:

$$E_f = \{\langle \mathsf{s}_1, \mathsf{s}_2 \rangle \mid \mathsf{s}_1, \mathsf{s}_2 \in S \wedge f(\mathsf{s}_1) = f(\mathsf{s}_2)\}.$$

In the following, with an abuse of notation we will simply write $S_{/f}$ instead of $S_{/E_f}$.

Given a poset $\langle R, \sqsubseteq \rangle$, a set D and two functions $f, g \in [D \to R]$, then the order $\sqsubseteq$ can be lifted to the point-wise order $\dot{\sqsubseteq}$ on functions:

$$f \dot{\sqsubseteq} g \iff \forall \mathsf{d} \in D. f(\mathsf{d}) \sqsubseteq g(\mathsf{d}).$$

If $\langle D, \sqsubseteq \rangle$ and $\langle R, \preceq \rangle$ are complete lattices, then we say that a function $f \in [D \to R]$

– is monotonic if it preserves the order of the elements:

$$\forall \mathsf{d}_1, \mathsf{d}_2 \in D.\ \mathsf{d}_1 \sqsubseteq \mathsf{d}_2 \Longrightarrow f(\mathsf{d}_1) \preceq f(\mathsf{d}_2);$$

– is a join-morphism if it preserves least upper bounds:

$$\forall \mathsf{d}_1, \mathsf{d}_2 \in D.\ f(\mathsf{d}_1 \sqcup \mathsf{d}_2) = f(\mathsf{d}_1) \curlyvee f(\mathsf{d}_2)$$

– is a *complete* join-morphism if it preserves least upper bounds for arbitrary subsets of D:

$$\forall \{\mathsf{d}_i\} \subseteq D.\ f\left(\bigsqcup_i \mathsf{d}_i\right) = \bigvee_i f(\mathsf{d}_i)$$

– is continuous if it preserves the least upper bound of increasing chains:

$$\forall \{\mathsf{d}_i\} \subseteq D.\ \mathsf{d}_0 \sqsubseteq \mathsf{d}_1 \sqsubseteq \mathsf{d}_2 \ldots \Longrightarrow f\left(\bigsqcup_i \mathsf{d}_i\right) = \bigvee_i f(\mathsf{d}_i)$$

Similarly, a function f is said to be a meet-morphism if it preserves the greatest lower bound of two elements and a complete meet-morphism if it preserves the greatest lower bound for any subset of a complete lattice D.

Given a set D and a function $f \in [D \to D]$, a fixpoint of f is an element $\mathsf{d} \in D$ such that $f(\mathsf{d}) = \mathsf{d}$. If f is defined over a partial order $\langle D, \sqsubseteq \rangle$, then an element $\mathsf{d} \in D$ is:

– a pre-fixpoint if $\mathsf{d} \sqsubseteq f(\mathsf{d})$;

– a post-fixpoint if $f(\mathsf{d}) \sqsubseteq \mathsf{d}$;

– the least fixpoint if $\mathsf{d} = f(\mathsf{d})$ and $\forall \mathsf{d}' \in D.\ \mathsf{d}' = f(\mathsf{d}') \Longrightarrow \mathsf{d} \sqsubseteq \mathsf{d}'$;

– the greatest fixpoint if $\mathsf{d} = f(\mathsf{d})$ and $\forall \mathsf{d}' \in D.\ \mathsf{d}' = f(\mathsf{d}') \Longrightarrow \mathsf{d}' \sqsubseteq \mathsf{d}$.

Given a function f defined over a poset $\langle D, \sqsubseteq \rangle$ and an element $\mathsf{d} \in D$, we denote with $\mathrm{lfp}^{\sqsubseteq}_{\mathsf{d}} f$, the least fixpoint of f w.r.t. the order $\sqsubseteq$ larger than d, if it exists. Sometimes, when the order and the element are clear from the context, we will simply write $\mathrm{lfp}\, f$. The definition of $\mathrm{gfp}^{\sqsubseteq}_{\mathsf{d}} f$ is dual.

A main result of Tarski [104] is that a monotonic function defined over a complete lattice admits a least and greatest fixpoint:

THEOREM 2.1 (FIXPOINT THEOREM, TARSKI [104]) *Let* $\langle D, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ *be a complete lattice and let* $f \in [D \to D]$ *be a monotonic function. Then the set* $F = \{\mathsf{d} \in D \mid f(\mathsf{d}) = \mathsf{d}\}$ *is a non-empty complete lattice w.r.t the order* $\sqsubseteq$*. Furthermore*

$$\mathrm{lfp}_{\bot}^{\sqsubseteq} f = \sqcap\{\mathsf{d} \in D \mid f(\mathsf{d}) \sqsubseteq \mathsf{d}\}$$
$$\mathrm{gfp}_{\bot}^{\sqsubseteq} f = \sqcup\{\mathsf{d} \in D \mid f(\mathsf{d}) \sqsupseteq \mathsf{d}\}.$$

The result of the theorem is not constructive and an alternative characterization of the least fixpoint for monotonic functions defined over a complete lattice can be given using the following theorem:

THEOREM 2.2 (TRANSFINITE ITERATIONS, COUSOT & COUSOT [30]) *Let* $\langle D, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ *be a complete lattice,* $f \in [D \to D]$ *be a monotonic function and let the transfinite iteration sequence be defined as:*

$$\begin{aligned} f^0(\bot) &= \bot \\ f^{\delta+1}(\bot) &= f(f^{\delta}(\bot)) && \text{for successor ordinals } \delta \\ f^{\lambda}(\bot) &= \bigsqcup_{\delta \le \lambda} f^{\delta}(\bot) && \text{for limit ordinals } \lambda. \end{aligned} \tag{2.1}$$

Then the increasing sequence $f^{\delta}, \delta \in \mathbb{O}$*, is ultimately stationary at rank* $\rho \in \mathbb{O}$ *and converges to* $f^{\rho} = \mathrm{lfp}_{\bot}^{\sqsubseteq} f$*.*

The result can be generalized in order to obtain $\mathrm{lfp}_{\mathsf{d}}^{\sqsubseteq} f$ for any $\mathsf{d} \in D$ such that $\mathsf{d} \sqsubseteq f(\mathsf{d})$ [30]. It suffices to change the base case of (2.1) with $f^0(\bot) = \mathsf{d}$.

2.1.3 Traces

DEFINITION 2.1 (TRACES) *Given a set* Σ *of states and an* $\Omega \notin \Sigma$*, a trace* τ *is a function* $\tau \in [\mathbb{N} \to \Sigma \cup \{\Omega\}]$ *which respects the prefix condition:*

$$\forall n \in \mathbb{N}.\ \tau(n) = \Omega \Longrightarrow \forall i > n.\ \tau(i) = \Omega.$$

Roughly, if a trace is undefined for an $n \in \mathbb{N}$, then it is undefined for all the successors of n too. We say that a trace τ is finite if $\exists n \in \mathbb{N}.\tau(n) = \Omega$. If not we say that it is infinite. The sets of finite traces over Σ is denoted by $\mathcal{T}(\Sigma)$.

DEFINITION 2.2 (LENGTH OF A TRACE, len) *The function* $\mathrm{len} \in [\mathcal{T}(\Sigma) \to \mathbb{N}]$ *is defined as*

$$\mathrm{len} = \lambda\tau.\ \min\{n \in \mathbb{N} \mid \tau(n) = \Omega \Longrightarrow \forall i > n.\ \tau(i) = \Omega\}.$$

A trace can be isomorphically represented by a succession of states. If τ is finite then we can write $\tau = \sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \sigma_n$ where $n = \mathrm{len}(\tau) - 1$ and $\forall i \in [0..n].\ \sigma_i = \tau(i)$. The empty trace is denoted by ϵ and a trace of length one is just a state $\sigma \in \Sigma$. Given a trace τ, we assume that $\tau \rightarrow \epsilon = \tau$.

Given a set of traces $T \subseteq \mathcal{T}(\Sigma)$, a $\tau \in T$ is a *maximal* trace of T if $\nexists \tau' \in T.\ (\forall i \in [0..\mathrm{len}(\tau) - 1].\ \tau(i) = \tau'(i)) \wedge (\mathrm{len}(\tau) < \mathrm{len}(\tau'))$.

Traces can be labeled. Let us consider a set of states in the form of $\Sigma = \Sigma' \times \mathfrak{L}$, where $\mathfrak{L}$ is a set of labels that contains the empty label ϵ. Then, a trace $\tau \in \mathcal{T}(\Sigma)$ in the form of

$$\tau = \langle \sigma_0, \epsilon \rangle \rightarrow \langle \sigma_1, \ell_1 \rangle \rightarrow \langle \sigma_2, \ell_2 \rangle \ldots$$

can be rewritten as a *labeled* trace:

$$\tau = \sigma_0 \xrightarrow{\ell_1} \sigma_1 \xrightarrow{\ell_2} \sigma_1 \ldots .$$

Sometimes, with an abuse of notation we tacitly assume the set of labels $\mathfrak{L}$ is given, so that we write $\mathcal{T}(\Sigma')$ instead of the more verbose $\mathcal{T}(\Sigma' \times \mathfrak{L})$.

Given a program $\mathtt{P}$, the associated transition relation $\xrightarrow{\mathtt{P}} \subseteq \Sigma \times \Sigma$, and a set of initial states S_0, the *partial traces* semantics of $\mathtt{P}$ can be expressed as a fixpoint:

Theorem 2.3 (Fixpoint Partial Traces Semantics, [25, 31]) *Let Σ be a set of states, $\xrightarrow{\mathtt{P}} \subseteq \Sigma \times \Sigma$ be the transition relation associated with a program $\mathtt{P}$, $S_0 \subseteq \Sigma$ be a set of initial states and $F \in [\mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\mathcal{T}(\Sigma)) \rightarrow \mathcal{P}(\mathcal{T}(\Sigma))]$ be*

$$F(S_0) = \lambda X.S_0 \cup \{\sigma_0 \rightarrow \ldots \sigma_n \rightarrow \sigma_{n+1} \mid \sigma_0 \rightarrow \ldots \rightarrow \sigma_n \in X \wedge \sigma_n \xrightarrow{\mathtt{P}} \sigma_{n+1}\}.$$

Then the partial trace semantics of $\mathtt{P}$, $\mathbb{tr}[\![\mathtt{P}]\!] \in [\mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\mathcal{T}(\Sigma))]$, is:

$$\mathbb{tr}[\![\mathtt{P}]\!](S_0) = \mathrm{lfp}^{\subseteq}_{\emptyset} F(S_0) = \bigcup_{i \leq \omega} F^i(S_0).$$

With a slightly abuse of notation, in the following we often write $\rightarrow$ for $\xrightarrow{\mathtt{P}}$, so that in particular we do not differentiate between the transition relation $\xrightarrow{\mathtt{P}}$ and the $\rightarrow$ used for denoting traces.

2.2 Abstract Interpretation

The abstract interpretation theory, due to P. Cousot [25, 29, 31], formalizes the approximation correspondence between the concrete semantics $\mathbb{s}[\![\mathtt{P}]\!]$ of a syntactically correct program $\mathtt{P}$ and an abstract semantics $\bar{\mathbb{s}}[\![\mathtt{P}]\!]$ which is a safe approximation on the concrete semantics.

2.2.1 Galois Connections

The concrete semantics belongs to a concrete semantic domain D which is a partially ordered set $\langle D, \sqsubseteq \rangle$. In such a setting, the partial order $\sqsubseteq$ formalizes the loss of information, e.g. the logical implication. The abstract semantics also belongs to a partial order $\langle \bar{D}, \bar{\sqsubseteq} \rangle$, which is ordered by the abstract version $\bar{\sqsubseteq}$ of the concrete approximation order $\sqsubseteq$. We use the notation to tag the abstract counterparts for concrete entities with an over-line. For instance, $\bar{D}$ and $\bar{\sqsubseteq}$ are the abstract counterparts for the concrete domain D and the order $\sqsubseteq$.

Definition 2.3 (Galois Connections, [29]) *Let $\langle D, \sqsubseteq \rangle$ and $\langle \bar{D}, \bar{\sqsubseteq} \rangle$ be two partial orders and let $\alpha \in [D \to \bar{D}]$ and $\gamma \in [\bar{D} \to D]$. If*

$$\forall \mathsf{d} \in D. \forall \bar{\mathsf{d}} \in \bar{D}.\ \alpha(\mathsf{d}) \bar{\sqsubseteq} \bar{\mathsf{d}} \iff \mathsf{d} \sqsubseteq \gamma(\bar{\mathsf{d}}) \tag{2.2}$$

then we say that $\langle \alpha, \gamma \rangle$ is a Galois connection and we denote it as

$$\langle D, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{D}, \bar{\sqsubseteq} \rangle.$$

Theorem 2.4 ([29]) *Let $\langle D, \sqsubseteq \rangle$ and $\langle \bar{D}, \bar{\sqsubseteq} \rangle$ be two partial orders and let $\alpha \in [D \to \bar{D}]$ and $\gamma \in [\bar{D} \to D]$ be two maps such that:*

1. *α and γ are monotonic functions*
2. *$\forall \mathsf{d} \in D.\ \mathsf{d} \sqsubseteq \gamma \circ \alpha(\mathsf{d})$*
3. *$\forall \bar{\mathsf{d}} \in \bar{D}.\ \alpha \circ \gamma(\bar{\mathsf{d}}) \bar{\sqsubseteq} \bar{\mathsf{d}}$.*

Then $\langle D, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{D}, \bar{\sqsubseteq} \rangle$.

Example 2.1 (States Abstraction, α_Σ) Let Σ be a set of states. Then

$$\langle \mathcal{P}(\mathcal{T}(\Sigma)), \subseteq, \emptyset, \mathcal{T}(\Sigma), \cup, \cap \rangle \underset{\alpha_\Sigma}{\overset{\gamma_\Sigma}{\leftrightarrows}} \langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle,$$

where the abstraction and the concretization functions are defined as follows:

$$\alpha_\Sigma(T) = \{\sigma \in \Sigma \mid \exists \tau \in T.\ \exists i.\ \tau(i) = \sigma\}$$
$$\gamma_\Sigma(S) = \{\tau \in \mathcal{T}(\Sigma) \mid \forall i.\ \tau(i) \neq \Omega \implies \tau(i) \in S\}.$$

Example 2.2 (Final States Abstraction, $\alpha_\dashv$) Let Σ be a set of states. Then

$$\langle \mathcal{P}(\mathcal{T}(\Sigma)), \subseteq, \emptyset, \mathcal{T}(\Sigma), \cup, \cap \rangle \underset{\alpha_\dashv}{\overset{\gamma_\dashv}{\leftrightarrows}} \langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle,$$

where the abstraction and the concretization functions are defined as follows:

$$\alpha_\dashv(T) = \{\sigma \in \Sigma \mid \exists \tau \in T.\ \tau \text{ is maximal } \wedge \tau(\text{len}(\tau) - 1) = \sigma\}$$
$$\gamma_\dashv(S) = \{\tau \in \mathcal{T}(\Sigma) \mid \exists n \geq 0.\ \tau(n) \in S \wedge \tau(n+1) = \Omega\}.$$

Galois connections enjoy several properties [33]:

LEMMA 2.1 (COMPOSITION) *If* $\langle D, \sqsubseteq\rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \bar{D}_1, \bar{\sqsubseteq}_1\rangle$ *and* $\langle \bar{D}_1, \bar{\sqsubseteq}_1\rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \bar{D}, \bar{\sqsubseteq}\rangle$ *then* $\langle D, \sqsubseteq\rangle \underset{\alpha_2\circ\alpha_1}{\overset{\gamma_1\circ\gamma_2}{\leftrightarrows}} \langle \bar{D}, \bar{\sqsubseteq}\rangle$.

LEMMA 2.2 (UNICITY OF THE ADJOINT) *Let* $\langle D, \sqsubseteq\rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \bar{D}, \bar{\sqsubseteq}\rangle$ *and* $\langle D, \sqsubseteq\rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \bar{D}, \bar{\sqsubseteq}\rangle$. *Then* $\alpha_1 = \alpha_2$ *if and only if* $\gamma_1 = \gamma_2$.

LEMMA 2.3 (DETERMINATION OF THE ADJOINT) *If* $\langle D, \sqsubseteq\rangle$ *and* $\langle \bar{D}, \bar{\sqsubseteq}\rangle$ *are complete lattices and if* $\langle D, \sqsubseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{D}, \bar{\sqsubseteq}\rangle$ *then*

$$\alpha(\mathsf{d}) = \bar{\sqcap}\{\bar{\mathsf{d}} \in \bar{D} \mid \mathsf{d} \sqsubseteq \gamma(\bar{\mathsf{d}})\} \qquad \textit{and}$$
$$\gamma(\bar{\mathsf{d}}) = \sqcup\{\mathsf{d} \in D \mid \alpha(\mathsf{d}) \bar{\sqsubseteq} \bar{\mathsf{d}}\}.$$

LEMMA 2.4 (PRESERVATION OF BOUNDS) *If* $\langle D, \sqsubseteq\rangle$ *and* $\langle \bar{D}, \bar{\sqsubseteq}\rangle$ *are complete lattices and if* $\langle D, \sqsubseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{D}, \bar{\sqsubseteq}\rangle$ *then* α *is a complete join-morphism and* γ *is a complete meet-morphism.*

LEMMA 2.5 (LIFTING TO THE HIGHER-ORDER) *Let* $\langle D_i, \sqsubseteq_i\rangle \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} \langle \bar{D}_i, \bar{\sqsubseteq}_i\rangle$, $\langle \bar{D}_o, \bar{\sqsubseteq}_o\rangle \underset{\alpha_o}{\overset{\gamma_o}{\leftrightarrows}} \langle \bar{D}_o, \bar{\sqsubseteq}_o\rangle$. *Furthermore, let* $[D_i \xrightarrow{m} D_o]$ *and* $[\bar{D}_i \xrightarrow{m} \bar{D}_o]$ *be sets of monotonic functions. Then*

$$\langle [D_i \xrightarrow{m} D_o], \dot{\sqsubseteq}\rangle \underset{\dot{\alpha}}{\overset{\dot{\gamma}}{\leftrightarrows}} \langle [\bar{D}_i \xrightarrow{m} \bar{D}_o], \dot{\bar{\sqsubseteq}}\rangle,$$

where

$$\dot{\alpha}(f) = \lambda\bar{\mathsf{v}}.\ \alpha_o \circ f \circ \gamma_i(\bar{\mathsf{v}}) \qquad \textit{and}$$
$$\dot{\gamma}(\bar{f}) = \lambda\mathsf{v}.\ \gamma_o \circ \bar{f} \circ \alpha_i(\mathsf{v}).$$

THEOREM 2.5 (LATTICE OF ABSTRACT INTEPRETATIONS, [25, 31]) *Let* $\langle D, \sqsubseteq\rangle$ *be a complete lattice. Furthermore, let*

$$A(D) = \{\langle \bar{D}, \bar{\sqsubseteq}\rangle \mid \exists\langle\alpha, \gamma\rangle.\ \langle D, \sqsubseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{D}, \bar{\sqsubseteq}\rangle\}$$

and the order $\sqsubseteq_A$ *on* $A(D)$ *be defined as*

$$\sqsubseteq_A = \{\langle\langle \bar{D}_1, \bar{\sqsubseteq}_1\rangle, \langle \bar{D}_1, \bar{\sqsubseteq}_2\rangle\rangle \mid \exists\langle\alpha, \gamma\rangle.\ \langle \bar{D}_1, \bar{\sqsubseteq}_1\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{D}_1, \bar{\sqsubseteq}_2\rangle\}.$$

Then, $\langle A(D), \sqsubseteq_A\rangle$ *is a complete lattice. In particular, the least element of* $A(D)$ *is* $\langle D, \sqsubseteq\rangle$.

DEFINITION 2.4 (REDUCED PRODUCT, [31]) *Under the hypotheses of the previous theorem, let* $\sqcap_A$ *be the meet operator on* $A(D)$. *Then, the reduced product,* $\otimes \in [A(D) \times A(D) \to A(D)]$, *is defined as*

$$\otimes = \lambda\langle D_1, D_2\rangle.\ D_1 \sqcap_A D_2.$$

2.2.2 Fixpoint Approximation

The concrete and abstract semantics of a program $\mathtt{P}$ are usually given in a fixpoint form as $\mathbb{s}[\![\mathtt{P}]\!] = \mathrm{lfp}\,\mathbb{t}[\![\mathtt{P}]\!]$ and $\bar{\mathbb{s}}[\![\mathtt{P}]\!] = \mathrm{lfp}\,\bar{\mathbb{t}}[\![\mathtt{P}]\!]$, where the semantic transformers $\mathbb{t}[\![\mathtt{P}]\!]$ and $\bar{\mathbb{t}}[\![\mathtt{P}]\!]$ are monotonic functions respectively on $[D \to D]$ and $[\bar{D} \to \bar{D}]$. If the concrete and abstract transformer satisfy the *local commutative condition* $\mathbb{t}[\![\mathtt{P}]\!] \circ \gamma \sqsubseteq \gamma \circ \bar{\mathbb{t}}[\![\mathtt{P}]\!]$ then the next theorem ensures the soundness of the abstract semantics:

THEOREM 2.6 (FIXPOINT TRANSFER, [31]) *Let $\langle D, \sqsubseteq\rangle$ and $\langle \bar{\mathsf{D}}, \bar{\sqsubseteq}\rangle$ be complete lattices. Let $\mathbb{t}[\![\mathtt{P}]\!]$ and $\bar{\mathbb{t}}[\![\mathtt{P}]\!]$ be monotonic functions defined respectively on $[D \to D]$ and $[\bar{D} \to \bar{D}]$. If $\langle D, \sqsubseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{\mathsf{D}}, \bar{\sqsubseteq}\rangle$ and if $\mathbb{t}[\![\mathtt{P}]\!] \circ \gamma \sqsubseteq \gamma \circ \bar{\mathbb{t}}[\![\mathtt{P}]\!]$ then*

$$\mathbb{s}[\![\mathtt{P}]\!] = \mathrm{lfp}\,\mathbb{t}[\![\mathtt{P}]\!] \sqsubseteq \gamma(\mathrm{lfp}\,\bar{\mathbb{t}}[\![\mathtt{P}]\!]) = \gamma(\bar{\mathbb{s}}[\![\mathtt{P}]\!]).$$

The abstract fixpoint $\mathrm{lfp}\ \bar{\mathbb{t}}[\![\mathtt{P}]\!]$ is computed as the limit of the iteration sequence (2.1). If the abstract domain $\bar{D}$ enjoys the ACC condition then the computation is guaranteed to terminate. If not the sequence convergence must be assured using a widening operator [29]. Informally a widening is a kind of join for which every increasing sequence is stationary after a finite number of steps and that *extrapolates* the sequence limit. Formally:

DEFINITION 2.5 (WIDENING, [29]) *A widening $\bar{\nabla} \in [\bar{D} \times \bar{D} \to \bar{D}]$ is an operator such that*

$$\forall \bar{\mathsf{d}}_1, \bar{\mathsf{d}}_2 \in \bar{D}.\ \bar{\mathsf{d}}_1 \bar{\sqsubseteq} \bar{\mathsf{d}}_1 \bar{\nabla} \bar{\mathsf{d}}_2 \text{ and } \bar{\mathsf{d}}_2 \bar{\sqsubseteq} \bar{\mathsf{d}}_1 \bar{\nabla} \bar{\mathsf{d}}_2$$

and for all increasing chains $\bar{\mathsf{d}}_0 \bar{\sqsubseteq} \bar{\mathsf{d}}_1 \bar{\sqsubseteq} \ldots \bar{\sqsubseteq} \bar{\mathsf{d}}_{i+1} \ldots$ the increasing chain defined by

$$\bar{\mathsf{y}}_0 = \bar{\mathsf{d}}_0, \bar{\mathsf{y}}_1 = \bar{\mathsf{y}}_0 \bar{\nabla} \bar{\mathsf{d}}_1 \ldots, \bar{\mathsf{y}}_{i+1} = \bar{\mathsf{y}}_i \bar{\nabla} \bar{\mathsf{d}}_{i+1} \ldots$$

is not strictly *increasing.*

A widening operator can be employed to force the convergence of the computation of the abstract semantics to a post-fixpoint. In fact the following result holds:

THEOREM 2.7 (ITERATIONS WITH WIDENING, [29]) *The following iteration sequence*

$$X^0 = \bar{\bot}$$
$$X^{i+1} = \begin{cases} X^i & \text{if } \mathbb{t}[\![\mathtt{P}]\!](X^i) \bar{\sqsubseteq} X^i \\ X^i \bar{\nabla} \mathbb{t}[\![\mathtt{P}]\!](X^i) & \text{otherwise} \end{cases}$$

is ultimately stationary and its limit $\bar{\mathsf{X}}$ *is a post-fixpoint for* $\bar{\mathfrak{t}}[\![\mathtt{P}]\!]$*:* $\mathrm{lfp}^{\bar{\sqsubseteq}}_{\bar{\perp}}\bar{\mathfrak{t}}[\![\mathtt{P}]\!] \bar{\sqsubseteq} \bar{\mathsf{X}}$*. Thus, it is a sound approximation of the semantics:*

$$\mathrm{lfp}^{\sqsubseteq}_{\perp}\mathfrak{t}[\![\mathtt{P}]\!] \sqsubseteq \gamma(\mathrm{lfp}^{\bar{\sqsubseteq}}_{\bar{\perp}}\bar{\mathfrak{t}}[\![\mathtt{P}]\!]) \sqsubseteq \gamma(\bar{\mathsf{X}}).$$

Several variants of the above theorem are possible. For example one may use a different widening at each iterate or even define a widening that depends upon all the previous iterations [34].

2.2.3 Chaotic and Asynchronous Iterations

Let us consider a monotonic function $F \in [D_1 \times \ldots D_n \to D_1 \times \cdots \times D_n]$ where each $\langle D_i, \sqsubseteq_i \rangle$, $i \in [1..n]$ is a complete lattice. Then, the fixpoint equation $X = F(X)$, $X \in D_1 \times \cdots \times D_n$ can be decomposed into the system of equations [25, 24]:

$$\{X_i = F_i(X_1, \ldots, X_n) \mid i \in [1..n], X_i \in D_i\}.$$

Let $J \subseteq [1..n]$. We denote by $F_J \in [D_1 \times \cdots \times D_n \to D_1 \times \ldots D_n]$ the function $F_J(X_1, \ldots X_n) = (Y_1, \ldots Y_n)$ where for all $i \in [1..n]$:

$$Y_i = \begin{cases} F_i(X_1, \ldots X_n) & \text{if } i \in J \\ X_i & \text{otherwise.} \end{cases}$$

Definition 2.6 (Ascending Chaotic Iterations, [25]) *Let* $C = \{J^\delta \subseteq [1..n] \mid \delta \in \mathbb{O}\}$ *such that*

$$\forall \delta \in \mathbb{O}. \forall i \in [1..n]. \exists J^\rho \in C.\ \delta \leq \rho \wedge i \in J^\rho.$$

Then the chaotic iteration sequence for the monotonic function $F \in [D_1 \times \ldots D_n \to D_1 \times \cdots \times D_n]$ *is defined as*

$$\begin{aligned} X^0 &= (\perp_1, \ldots \perp_n) \\ X^{\delta+1} &= F_{J^\delta}(X^\delta) && \text{for successor ordinals } \delta \\ X^\lambda &= \bigsqcup_{\delta \leq \lambda} X^\delta && \text{for limit ordinals } \lambda. \end{aligned} \tag{2.3}$$

where $\sqcup$ *is the join operator on* $D_1 \times \cdots \times D_n$*.*

Theorem 2.8 (Convergence of Chaotic Iterations, Cousot [25]) *Under the hypotheses of Definition 2.6, the sequence* $X^\delta, \delta \in \mathbb{O}$ *is ultimately stationary at rank* $\rho \in \mathbb{O}$ *and* $X^\rho = \mathrm{lfp}^{\sqsubseteq_1 \times \cdots \times \sqsubseteq_n}_{(\perp_1, \ldots \perp_n)} F$*.*

Definition 2.7 (Ascending Asynchronous Iterations, [25]) *Let* $C = \{J^\delta \in [1..n] \mid \delta \in \mathbb{O}\}$ *such that*

$$\forall \delta \in \mathbb{O}. \forall i \in [1..n]. \exists J^\rho \in C.\ \delta \leq \rho \wedge i = J^\rho.$$

Let $D = \{S^\delta \mid \delta \in \mathbb{O}, S^\delta \in \mathbb{O}^n\}$ *such that for any* $S^\delta \in D$*:*

- $\forall i \in [1..n]. \forall \delta \in \mathbb{O}.\ S^\delta_{(i)} < \delta$
- $\forall i \in [1..n]. \forall \delta \in \mathbb{O}. \exists \rho \in \mathbb{O}. \forall \beta \geq \rho.\ \delta \leq S^\beta_{(i)}$
- $\forall \lambda, \delta \in \mathbb{O}. \lambda < \delta \wedge \lambda$ *is a limit ordinal* $\Longrightarrow \forall i \in [1..n].\ \lambda \leq S^\delta_{(i)}$.

Then the ascending asynchronous iteration sequence for the monotonic function $F \in [D_1 \times \cdots \times D_n \to D_1 \times \cdots \times D_n]$ *is defined as*

$$\begin{aligned} X^0 &= (\bot_1, \ldots \bot_n) \\ X^{\delta+1} &= F_{J^\delta}(X^{S^\delta_{(1)}}_{(1)}, \ldots X^{S^\delta_{(n)}}_{(n)}) && \text{for successor ordinals } \delta \\ X^\lambda &= \bigsqcup_{\delta \leq \lambda} X^\delta && \text{for limit ordinals } \lambda, \end{aligned}$$

where $\sqcup$ *is the join operator on* $D_1 \times \cdots \times D_n$*.*

Theorem 2.9 (Convergence of Asynchronous Iterations, Cousot [25]) *Under the hypotheses of Definition 2.7, the sequence* $X^\delta, \delta \in \mathbb{O}$ *is ultimately stationary at rank* $\rho \in \mathbb{O}$ *and* $X^\rho = \mathrm{lfp}^{\sqsubseteq_1 \times \cdots \times \sqsubseteq_n}_{(\bot_1, \ldots \bot_n)} F$*.*

Chapter 3

Concrete Semantics

> Semantics is not a device for establishing that everyone except the speaker and his friends is speaking nonsense.
>
> Alfred Tarski (1944)

In this chapter we introduce the concrete semantics of object-oriented programs. The concrete semantics describes the properties of the execution of programs. The goal of a static analysis is to provide an effective computable approximation of the concrete semantics [34]. Therefore, the first step of a static analysis is the definition of a concrete semantics.

We define two semantics. The first one is a trace semantics for a *whole* object-oriented program and the second one is a trace semantics for a *single* class. We study the relation between the two semantics, and in particular we show the soundness and the completeness of the class semantics w.r.t. the whole-program semantics.

3.1 Semantics of Object-oriented Languages in Literature

Several authors proposed different approaches to the semantics of object-oriented languages. We will review some of these semantics shortly, and we explain why they do not fulfill our needs. We considered approaches based on types [16], object calculi [62, 2], Abstract State Machines [100] and denotational semantics [64].

3.1.1 Types

Cardelli introduced in [16] the view of *objects as records*: objects are essentially records with possibly functional components (the object's methods). Message passing is achieved by field selection and inheritance deals with the cardinality and the type of the record fields. Therefore, in the object-as-record model objects are record values, classes are record generators and message passing is field selection.

The great advantage of the *objects as records* approach is its clarity and elegance. Unfortunately, it is not suitable for modeling object aliasing. In fact, in such a view two objects are the same if their fields have the same values. Therefore, the concept of *object identity*, and hence of object aliasing, cannot be captured. As a consequence, such a formalism cannot be used for modeling mainstream object-oriented languages.

3.1.2 Object Calculi

Object calculi are formalisms at the same level of abstraction as λ-calculus, but based on objects rather than functions. There is a wide spectrum of object calculi, as for example the Featherweight Java [62] or ζ-calculus [2]. Object calculi are meant to capture some properties of realistic programming languages, while abstracting away from others. For example, Featherweight Java is an object calculus that abstracts away from the store as its main concern is to prove the type-soundness of a subset of the Java programming language. Similarly, ζ-calculus presents a low-level mechanism for object updating and in [1] it has been extended with imperative features.

We have chosen not to use existing object calculi as, in general, they put themselves into a level of abstraction that is too abstract to capture the properties we are interested in. For instance, the notions of program points or call stack are missing. Furthermore, the formulation of these calculi as rewriting systems hides the underlying fixpoint computation. As a consequence, the adoption of one of such calculi would have required some additional work in order to make the fixpoint explicit.

3.1.3 Abstract State Machines

Abstract State Machines (ASM) are a formalism for modeling complex systems that has been widely used in the software engineering field [11]. It is based on the use of a transition relation specified as a set of conditional statements. The conditions are expressed using first order logic and naive set theory. Stärk et al. [100] used them to formalize the semantics of the full Java language. However, the use of the ASM as a concrete semantics has two

main drawbacks. First, the ASM notation has the tendency to hide low-level details, by making wide use of macros. While this may be an advantage to the casual reader, it is a drawback for the design of precise yet sound static analyses. Second, the program computation is hidden in the ASM transition relation, and the fixpoint computation is not explicitly stated. As a consequence, this formalism is inadequate to express e.g. program-wide invariant properties.

3.1.4 Denotational Semantics

Denotational semantics is well-suited for modeling object-oriented languages. In fact, object's self application and inheritance can be smartly expressed as fixpoints on suitable domains [64, 23]. Moreover, in such a setting it is straightforward to consider a domain composed by an environment, i.e. a map from variables to addresses, and a store, i.e. a map from addresses to values, so that object aliasing can be naturally expressed. However, it can be shown that generally denotational semantics is an abstraction of a trace-based operational semantics [28]. Intuitively, it abstracts away the history of computations, by considering input-output functions. As a consequence, we prefer to start directly with a small-step operational semantics rather than with one of its abstractions.

3.2 Whole-Program Trace Semantics

The trace semantics models the program computation by a set of state sequences. In this section we define the syntax of a class and of an object-oriented program. Then we define the semantic domains and the trace semantics of a whole object-oriented program.

3.2.1 Syntax

A class is a description for a set of objects, instances of the class. The class source is provided by the programmer, who specifies the class constructor, the fields and the methods. Therefore, the class syntax can be modeled as a triplet:

DEFINITION 3.1 (CLASS) *A class* $\mathtt{A}$ *is a triplet* $\langle \mathtt{init}, \mathtt{F}, \mathtt{M} \rangle$ *where* $\mathtt{init}$ *is the class constructor,* $\mathtt{F}$ *is a set of variables and* $\mathtt{M}$ *is a set of function definitions.*

We denote by $\mathcal{C}$ the set of all the classes. It is worth noting that for the sake of generality in the definition above we do not require to have typed fields or methods. Moreover we assume that all the class fields are protected.

This is done to simplify the exposition, and it does not cause any loss of generality: in fact any access to a field $\mathtt{f}$ can be simulated by a couple of methods $\mathtt{set\_f}/\mathtt{get\_f}$. Additionally we also assume that a class has just a single constructor, the generalization to an arbitrary number of constructors being straightforward.

An object-oriented program is a set of classes. In particular, it is made up by a main class and a library:

Definition 3.2 (Object-oriented Program) *An object-oriented program* $\mathtt{P}$ *is a pair* $\mathtt{P} = \langle \mathtt{A}_{\mathtt{main}}, \mathtt{L} \rangle$ *where* $\mathtt{A}_{\mathtt{main}} \in \mathcal{C}$ *is the main class and* $\mathtt{L} \subseteq \mathcal{C}$ *is the program library.*

With an abuse of notation, given an object-oriented program $\mathtt{P} = \langle \mathtt{A}_{\mathtt{main}}, \mathtt{L} \rangle$ in the following we write $\mathtt{A} \in \mathtt{P}$ if $\mathtt{A} = \mathtt{A}_{\mathtt{main}}$ or $\mathtt{A} \in \mathtt{L}$.

3.2.2 Semantic Domains

The first step in specifying the semantics is the definition of the semantic domains. Intuitively we need a domain that models that an object has its own identity and environment. Moreover, in order to be realistic, we need to express object aliasing.

In order to fulfill the above requirements, we consider a domain whose elements are pairs of environments and stores: an environment is a map from variable names to memory addresses, and a store is a map from addresses to memory elements. In its turn, a memory element can be a primitive value or an environment. The object environment is stored in a certain memory location whose address represents the object identity. In such a setting, two distinct variables are aliases for an object if they reference the same object, i.e. the same memory address. We can formalize what said above:

Definition 3.3 (Environment, Env) *Let* $\mathtt{Var}$ *be a set of variables and let* $\mathsf{Addr} \subseteq \mathbb{N}$ *be a set of addresses. Then the set of environments is* $\mathsf{Env} = [\mathtt{Var} \to \mathsf{Addr}]$.

Definition 3.4 (Store, Store) *Let* Val *be a set of values such that* $\mathsf{Env} \subseteq \mathsf{Val}$. *Then the set of stores is* $\mathsf{Store} = [\mathsf{Addr} \to \mathsf{Val}]$.

Given a state $\rho \in \mathsf{Env} \times \mathsf{Store}$, $\rho = \langle \mathsf{e}, \mathsf{s} \rangle$ and a variable $\mathtt{x}$, we write $\rho(\mathtt{x})$ to mean $\mathsf{s}(\mathsf{e}(\mathtt{x}))$.

It is worth noting that in the above definition we require Env to be included in the possible memory values, so that given an object, i.e. a memory address, we can access its environment through the store. Moreover, in order

to be as generic as possible we put no further constraint on the values that can be stored in memory. Thus, for example the C++ memory model can be obtained once we have that $\mathsf{Addr} \subseteq \mathsf{Val}$.

We assume that a state contains some special variables, namely: the pointer to the current object, the current program counter [72, §3.5.1], the call stack [72, §3.6] and the method input value and the method return value, if any. This is summarized by the following definition:

DEFINITION 3.5 (SPECIAL VARIABLES) *Let* $\rho = \langle \mathsf{e}, \mathsf{s} \rangle \in \mathsf{Env} \times \mathsf{Store}$ *and* $\{\mathtt{this}, \mathtt{pc}, \mathtt{callStack}, \mathtt{inVal}, \mathtt{retVal}\} \subseteq \mathtt{Var}$. *Then*

- $\mathsf{e}(\mathtt{this})$ *is the pointer to the current object;*
- $\rho(\mathtt{pc})$ *is the program counter;*
- $\rho(\mathtt{callStack})$ *is the call stack;*
- $\rho(\mathtt{inVal})$ *is the input value of the current method;*
- $\rho(\mathtt{retVal})$ *is the value returned by the current method.*

In general, from the call stack it is possible to infer the current method (e.g. [72, §3.6]) and the height of the call stack. So, with an abuse of notation, we denote the current method of a state ρ with $\rho(\mathtt{curMethod})$ and the height of the call stack with $\rho(\mathtt{stackHeight})$.

3.2.3 Whole-Program Semantics

The semantics of an object-oriented program P is a set of traces that represents the executions of the program starting from a set of initial states R_0. According to Theorem 2.3, it can be expressed in fixpoint form as follows:

DEFINITION 3.6 (OBJECT-ORIENTED PROGRAM SEMANTICS, $\mathbb{w}[\![\cdot]\!]$) *Let* $\mathtt{P} = \langle \mathtt{A}_{\mathtt{main}}, \mathtt{L} \rangle$ *be an object-oriented program,* $\rightarrow \subseteq (\mathsf{Env} \times \mathsf{Store}) \times (\mathsf{Env} \times \mathsf{Store})$ *be a transition relation,* `main` *be a method of* $\mathtt{A}_{\mathtt{main}}$, $\mathtt{pc}_{\mathtt{in}}$ *be the entry point of* `main` *and* $R_0 \in \mathcal{P}(\mathsf{Env} \times \mathsf{Store})$ *be a set of program initial states, i.e. such that:*

$$\forall \rho_0 \in R_0.\ \rho_0(\mathtt{curMethod}) = \mathtt{main} \wedge \rho_0(\mathtt{pc}) = \mathtt{pc}_{\mathtt{in}}.$$

Then the semantics of P, $\mathbb{w}[\![\mathtt{P}]\!] \in [\mathcal{P}(\mathsf{Env} \times \mathsf{Store}) \rightarrow \mathcal{P}(\mathcal{T}(\mathsf{Env} \times \mathsf{Store}))]$ *is*

$$\mathbb{w}[\![\mathtt{P}]\!](R_0) = \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda X.\ R_0 \cup \{\rho_0 \rightarrow \ldots \rho_n \rightarrow \rho_{n+1} \mid \rho_{n+1} \in \mathsf{Env} \times \mathsf{Store} \wedge \rho_0 \rightarrow \ldots \rho_n \in X \wedge \rho_n \rightarrow \rho_{n+1}\}.$$

It is worth noting that for the sake of generality we do not explicitly define the transition relation $\rightarrow$ as it depends upon the particular considered object-oriented language.

The transition relation $\rightarrow$ (and the Theorem 2.3) can be used to express the semantics of the constructor and of the methods in fixpoint form as:

DEFINITION 3.7 (METHOD TRACE SEMANTICS, $\mathbb{w}_{\mathtt{m}}[\![\cdot]\!]$) *Let* $\mathtt{A} = \langle \mathtt{init}, \mathtt{F}, \mathtt{M} \rangle$, $\mathtt{m}$ *be* `init` *or* $\mathtt{m} \in \mathtt{M}$. *Let* $\mathtt{pc_{in}}$ *and* $\mathtt{pc_{exit}}$ *be respectively the entry point and the exit point of* $\mathtt{m}$. *Furthermore, let* $\rightarrow \subseteq (\mathsf{Env} \times \mathsf{Store}) \times (\mathsf{Env} \times \mathsf{Store})$ *and let* $R_0 \in \mathcal{P}(\mathsf{Env} \times \mathsf{Store})$ *be a set of method initial states, i.e. :*

$$\forall \rho_0 \in R_0.\ \rho_0(\mathtt{pc}) = \mathtt{pc_{in}}.$$

Then, the trace semantics of $\mathtt{m}$, $\mathbb{w}_{\mathtt{m}}[\![\mathtt{m}]\!] \in [\mathcal{P}(\mathsf{Env} \times \mathsf{Store}) \rightarrow \mathcal{P}(\mathcal{T}(\mathsf{Env} \times \mathsf{Store}))]$ *is*

$$\begin{aligned}\mathbb{w}_{\mathtt{m}}[\![\mathtt{m}]\!](R_0) = \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda X.\ R_0 \cup \{\rho_0 \rightarrow \ldots \rho_n \rightarrow \rho_{n+1} \mid \rho_{n+1} \in \mathsf{Env} \times \mathsf{Store} \\ \wedge\, \rho_0 \rightarrow \ldots \rho_n \in X \wedge \rho_n \rightarrow \rho_{n+1}\} \\ \cup \{\rho_0 \rightarrow \ldots \rho_n \mid \rho_0 \rightarrow \ldots \rho_n \in X \wedge \\ \rho_0(\mathtt{stackHeight}) = \rho_n(\mathtt{stackHeight}) \wedge \\ \rho_n(\mathtt{curMethod}) = \mathtt{m} \wedge \rho_n(\mathtt{pc}) = \mathtt{pc_{exit}}\}.\end{aligned}$$

Roughly, the trace semantics of a method $\mathtt{m}$ is such that the maximal traces end with a state such that the program counter refers to the exit point of $\mathtt{m}$ and the height of the stack is the same as the entry point. This last condition implies that we consider the exit point of $\mathtt{m}$ at the *same* level of recursion as the input.

3.3 Class Trace Semantics

We define the semantics of a class as the set of all the semantics of its instances. In its turn, the semantics of an object is the set of traces that correspond to the evolution of the object internal state. In this section, we give the fixpoint characterization of the class semantics. The soundness results of the next chapters will be given with respect to this concrete semantics.

3.3.1 Constructor and Methods Semantics

The semantics of an object is given on the top of the semantics of the class constructor and of the class methods. In its turn, the semantics of the constructor and the methods are obtained as an abstraction of the trace semantics of Definition 3.7.

When a class is instantiated, for example by means of the `new` construct, the class constructor is called to set up the new object internal state. The context, that creates the object, passes to the constructor the actual parameters and the store. The constructor initializes the object fields and returns the new object environment and the modified store. Such a behavior is formalized by the following definition:

DEFINITION 3.8 (CONSTRUCTOR SEMANTICS, $\mathbb{i}[\![\cdot]\!]$) *Let* $\mathsf{D_{in}} \subseteq \mathsf{Val}$ *be the semantic domain for the input values,* $\mathsf{e_0}$ *be the initial environment,* $\mathsf{a_{in}}, \mathsf{a_{pc}}$ *be fresh memory addresses,* $\mathtt{pc}_{\mathtt{init}}$ *be the constructor entry point and* $\alpha_{\dashv}$ *the abstraction function defined in Example 2.2. Then the semantics of the class constructor,* $\mathbb{i}[\![\mathtt{init}]\!] \in [\mathsf{D_{in}} \times \mathsf{Store} \to \mathcal{P}(\mathsf{Env} \times \mathsf{Store})]$*, is*

$$\mathbb{i}[\![\mathtt{init}]\!] = \lambda(\mathsf{v_{in}}, \mathsf{s}).\ \textit{let}\ \rho_0 = \langle \mathsf{e_0}[\mathtt{inVal} \mapsto \mathsf{a_{in}}, \mathtt{pc} \mapsto \mathsf{a_{pc}}], \mathsf{s}[\mathsf{a_{in}} \mapsto \mathsf{v_{in}}, \mathsf{a_{pc}} \mapsto \mathtt{pc}_{\mathtt{init}}]\rangle \\ \textit{in}\ \alpha_{\dashv}(\mathbb{w}_{\mathtt{m}}[\![\mathtt{init}]\!](\{\rho_0\})).$$

Here are some remarks on the above definition. First, the semantics $\mathbb{i}[\![\mathtt{init}]\!]$ is essentially an abstraction of $\mathbb{w}_{\mathtt{m}}[\![\mathtt{init}]\!]$ in which just the initial state and the final states are retained. Second, the output is a set of pairs as in general the execution of the constructor may be non-deterministic. Third, the generalization for a tuple of input values is straightforward.

EXAMPLE 3.1 (CONSTRUCTOR SEMANTICS) Let us consider the C++ class in Figure 3.1. The class constructor creates a new environment consisting of two entries, one for `i` and the other for `b`. Then it sets the value of `i` and `b` to that of the actual parameters `i0` and `b0`. The semantics of the `Ex` constructor is

$$\mathbb{i}[\![\mathtt{Ex} :: \mathtt{Ex}()]\!] = \lambda(\mathsf{i_0}, \mathsf{b_0}, \mathsf{s}).\ \{\langle \mathsf{e_0}[\mathtt{i} \mapsto \mathsf{a_i}, \mathtt{b} \mapsto \mathsf{a_b}, \mathtt{pc} \mapsto \mathsf{a_{pc}}], \\ \mathsf{s}[\mathsf{a_i} \mapsto \mathsf{i_0}, \mathsf{a_b} \mapsto \mathsf{b_0}, \mathsf{a_{pc}} \mapsto 11]\rangle\},$$

where $\mathsf{e_0}$ is an environment, $\mathsf{a_i}$, $\mathsf{a_b}$ and $\mathsf{a_{pc}}$ are fresh memory locations and `11` is the exit point of the constructor.

Once an object has been created, the context can interact with it. When the context invokes a method, it passes some input values, the object environment and the store. In its turn, the method returns a value, the new object environment and the modified store. The behavior of the method can be formalized by the following definition:

DEFINITION 3.9 (METHOD SEMANTICS, $\mathtt{m}[\![\cdot]\!]$) *Let* $\mathsf{D_{in}}, \mathsf{D_{out}} \subseteq \mathsf{Val}$ *be the semantic domains for the input values and the output values,* $\mathsf{a_{in}}$ *and* $\mathsf{a_{pc}}$ *be fresh memory addresses,* `m` *be a method,* $\mathtt{pc}_{\mathtt{m}}$ *be the* `m` *entry point and* $\alpha_{\dashv}$

```
class Ex {

 private:
  int i;
  bool b;

 public:
  Ex(int i0, bool b0) {
    i = i0;
    b = b0;
  };

  void modify(int delta) {
    if(b) i += delta;
    else i -= delta;
  };

  bool* getPointer() {
    return &b;
  };

};
```

Figure 3.1: An example of a C++ class

be the abstraction function defined in Example 2.2. Then the semantics of a method $\mathtt{m}$, $\mathbb{m}[\![\mathtt{m}]\!] \in [\mathsf{D_{in}} \times \mathsf{Env} \times \mathsf{Store} \to \mathcal{P}(\mathsf{D_{out}} \times \mathsf{Env} \times \mathsf{Store})]$, *is*

$$\begin{aligned}\mathbb{m}[\![\mathtt{m}]\!] = \lambda(\mathsf{v_{in}}, \mathsf{e}, \mathsf{s}).\ &\mathit{let}\ \rho_0 = \langle \mathsf{e}[\mathtt{inVal} \mapsto \mathsf{a_{in}}, \mathtt{pc} \mapsto \mathsf{a_{pc}}], \mathsf{s}[\mathsf{a_{in}} \mapsto \mathsf{v_{in}}, \mathsf{a_{pc}} \mapsto \mathtt{pc}_{\mathtt{m}}]\rangle \\ &\mathit{in}\ \mathit{let}\ R_f = \alpha_{\dashv}(\mathbb{w}_{\mathbb{m}}[\![\mathtt{m}]\!](\{\rho_0\})) \\ &\mathit{in}\ \{\langle \rho_f(\mathtt{retValue}), \mathsf{e}_f, \mathsf{s}_f\rangle \mid \rho_f = \langle \mathsf{e}_f, \mathsf{s}_f\rangle \in R_f\}.\end{aligned}$$

Some remarks on the above definition. First, it is essentially an input/output abstraction of $\mathbb{w}_{\mathbb{m}}[\![\mathtt{m}]\!]$. Second, the execution of the method may present some kind of non-determinism. Third, in order to model procedural methods, i.e. methods that does not return any value, we assume to have a special *void* value $\phi \in \mathsf{D_{out}}$. Fourth, if $\mathsf{Addr} \subseteq \mathsf{D_{out}}$ then the method may expose a part of the object internal state to the context.

Example 3.2 (Method Semantics) The class in Figure 3.1 has two methods. The first, `modify`, adds or subtracts a given value to `i` depending whether `b` is true or not. It does not return any value to the caller and it

does not change the object environment. Hence its semantics is:

$$\mathbb{m}[\![\texttt{modify}]\!] = \lambda(\mathsf{v},\mathsf{e},\mathsf{s}).\ \{\langle \phi, \mathsf{e}, \mathsf{s}[\mathsf{a}_{\mathsf{pc}} \mapsto 16, \mathsf{e}(\mathtt{i}) \mapsto \text{if } \mathsf{s}(\mathsf{e}(\mathtt{b})) = \mathsf{true} \text{ then } \mathsf{s}(\mathsf{e}(\mathtt{i})) + \mathsf{v} \text{ else } \mathsf{s}(\mathsf{e}(\mathtt{i})) - \mathsf{v}]\rangle\}.$$

It is worth noting that we update the value of the program counter to the exit point of `modify`, i.e. $\mathsf{a}_{\mathsf{pc}} = 16$.

The method `getPointer` is a helper method that returns a pointer to a part of the object state, namely to `b`. Hence this method exposes a part of the object state but it does not modify neither the environment nor the store (except for the program counter). The semantics of `getPointer` is:

$$\mathbb{m}[\![\texttt{getPointer}]\!] = \lambda(\phi,\mathsf{e},\mathsf{s}).\ \{\langle \mathsf{e}(\mathtt{b}), \mathsf{e}, \mathsf{s}[\mathsf{a}_{\mathsf{pc}} \mapsto 20]\rangle\}.$$

3.3.2 Object Semantics

The semantics of an object is given by a set of traces. Intuitively, each trace represents a possible evolution history of the object. The first state represents the object just after its creation. Each further state is the result of an interaction between the object and a context. This interaction can happen in two ways:

1. the context invokes a method of the object; or
2. it modifies a memory location that is reachable from the object environment.

We refer to the former as *direct interaction* and to the latter as *indirect interaction*. We do not consider a third possible kind of interaction, namely the one in which the context accesses the object fields unless the object has revealed them before. For instance, this is the case of a context that may guess the memory addresses where the object fields are stored.

For each interaction with the context, we consider the environment and the store after the interaction, the eventual value returned by a method execution and the set of addresses that escapes from the object. Formally, we define the *interaction states* as

Definition 3.10 (Interaction States) *The set of interaction states is* $\Sigma = \mathsf{Env} \times \mathsf{Store} \times \mathsf{D}_{\mathsf{out}} \times \mathcal{P}(\mathsf{Addr})$.

The *initial interaction states* are the states reached just after the creation of a class, i.e. after the invocation of the class constructor:

Definition 3.11 (Initial States, $S_0\langle \mathsf{v}, \mathsf{s}\rangle$) *Let* $\mathsf{v} \in \mathsf{D_{in}}$ *be an input value,* $\mathsf{s} \in \mathsf{Store}$ *be a store and* o *be an instance of a class* `A`. *Then the set of initial states of* o *is:*

$$S_0\langle \mathsf{v}, \mathsf{s}\rangle = \{\langle \mathsf{e}', \mathsf{s}', \phi, \emptyset\rangle \mid \mathtt{i}[\![\mathtt{init}]\!](\mathsf{v}, \mathsf{s}) \ni \langle \mathsf{e}', \mathsf{s}'\rangle\}.$$

It is worth noting that we use ϕ to model the fact that the class constructor does not return any value. As a consequence, it does not expose any address to the context.

Example 3.3 (Initial States) Let us instantiate `Ex` in some store s with initial values $\mathsf{3}$ and true. Then, ignoring the special variables, the initial states are given by:

$$S_0\langle\langle \mathsf{3}, \mathsf{true}\rangle, \mathsf{s}\rangle = \{\langle \mathsf{e}', \mathsf{s}', \phi, \emptyset\rangle \mid \begin{array}{l} \mathsf{e}' = [\mathtt{i} \mapsto \mathsf{a_i}, \mathtt{b} \mapsto \mathsf{a_b}], \\ \mathsf{s}' = \mathsf{s}[\mathsf{a_i} \mapsto \mathsf{3}, \mathsf{a_b} \mapsto \mathsf{true}]\}. \end{array}$$

The successors of a state are given by a transition function next. Given a state σ the intuitive meaning of $\text{next}(\sigma)$ is to consider all the states that are result of an interaction of σ with a context. Thus, as there are two kinds of interaction, next will consist of two parts: one corresponding to the direct interactions, and the other to the indirect ones. Moreover, in the following we will be interested to know whether a state is a consequence of a direct or an indirect interaction. For that purpose we tag each successor of σ with a label. A label can either be a pair consisting of the name of the invoked method and the input value, or a special term κ to denote an action of the context. Thus, the set of labels is

Definition 3.12 (Transition Labels, Label) *The set of labels is defined as* $\mathsf{Label} = (\mathtt{M} \times \mathsf{D_{in}}) \cup \{\kappa\}$.

The transition function, $\text{next} \in [\Sigma \to \mathcal{P}(\Sigma \times \mathsf{Label})]$, is made up of two components: next_{dir} and next_{ind}.

The direct interaction happens when the context invokes a method of the object. The context may invoke *any* method with *any* input value. Therefore, the transition function for the direct interaction is

Definition 3.13 (Direct Interaction, next_{dir}) *Let* $\langle \mathsf{e}, \mathsf{s}, \mathsf{v}, \mathsf{Esc}\rangle \in \Sigma$ *be an interaction state. Then the function* $\text{next}_{\text{dir}} \in [\Sigma \to \mathcal{P}(\Sigma \times \mathsf{Label})]$ *is defined as:*

$$\begin{array}{rl} \text{next}_{\text{dir}}(\langle \mathsf{e}, \mathsf{s}, \mathsf{v}, \mathsf{Esc}\rangle) = \{\langle\langle \mathsf{e}', \mathsf{s}', \mathsf{v}', \mathsf{Esc}'\rangle, \langle \mathtt{m}, \mathsf{v_{in}}\rangle\rangle \mid & \mathtt{m} \in \mathtt{M}, \mathsf{v_{in}} \in \mathsf{D_{in}}, \\ & \mathtt{m}[\![\mathtt{m}]\!](\mathsf{v_{in}}, \mathsf{e}, \mathsf{s}) \ni \langle \mathsf{v}', \mathsf{e}', \mathsf{s}'\rangle, \\ & \mathsf{Esc}' = \mathsf{Esc} \cup \text{reachable}(\mathsf{v}', \mathsf{s}')\}. \end{array}$$

In the above definition we made use of the helper function reachable, that given an address v and a store s, determines all the memory addresses that are reachable from v. Intuitively, if a method returns an address, then the context can access it so that the context can arbitrarily modify the value of the store at that point. In that case we say that the object exposes v or that v escapes the scope of the object. For the sake of simplicity, we assume that the only way that an object has to expose a part of its state is by returning an address to the context. At its turn, the memory element $\mathsf{s}(\mathsf{v}) = \mathsf{v}'$ may be (the identity of) an object. Then $\mathsf{s}(\mathsf{v}')$ is its environment, and (the addresses of) all its fields escape. And so on recursively:

DEFINITION 3.14 (reachable) *The function* $\text{reachable} \in [(\mathsf{D}_{\mathsf{out}} \times \mathsf{Store}) \to \mathcal{P}(\mathsf{Addr})]$ *is recursively defined as follows:*

$$\begin{aligned}\text{reachable}(\mathsf{v},\mathsf{s}) = &\textit{if } \mathsf{v} \in \mathsf{Addr} \textit{ then}\\ &\{\mathsf{v}\} \cup \{\text{reachable}(\mathsf{e}'(\mathtt{f}),\mathsf{s}) \mid \exists \mathtt{B}.\ \mathtt{B} = \langle \mathtt{init}, \mathtt{F}, \mathtt{M}\rangle,\ \mathtt{f} \in \mathtt{F},\\ &\qquad \mathsf{s}(\mathsf{v}) \textit{ is an instance of } \mathtt{B},\ \mathsf{s}(\mathsf{s}(\mathsf{v})) = \mathsf{e}'\}\\ &\textit{else } \emptyset.\end{aligned}$$

EXAMPLE 3.4 (DIRECT INTERACTION) Let us apply the above definitions to determine the successors of the state $\sigma_0 \in S_0\langle\langle 3, \mathsf{true}\rangle, \mathsf{s}\rangle$. In that case, the context can arbitrarily decide to invoke one of the methods of `Ex`. Thus, the successors of σ_0 are:

$$\begin{aligned}\text{next}_{\text{dir}}(\sigma_0) = &\{\langle\langle \mathsf{e}, \mathsf{s}', \phi, \emptyset\rangle, \langle \mathtt{modify}, \mathsf{v}\rangle\rangle \mid \mathbb{m}[\![\mathtt{modify}]\!](\mathsf{v},\mathsf{e},\mathsf{s}) \ni \langle \phi, \mathsf{e}, \mathsf{s}'\rangle,\\ &\qquad \mathsf{s}' = \mathsf{s}[\mathsf{a}_{\mathsf{i}} \mapsto 3 + \mathsf{v}]\}\\ &\cup \{\langle\langle \mathsf{e}, \mathsf{s}, \mathsf{a}_{\mathsf{b}}, \{\mathsf{a}_{\mathsf{b}}\}\rangle, \langle \mathtt{getPointer}, \phi\rangle\rangle \mid\\ &\qquad \mathbb{m}[\![\mathtt{getPointer}]\!](\phi,\mathsf{e},\mathsf{s}) \ni \langle \mathsf{a}_{\mathsf{b}}, \mathsf{e}, \mathsf{s}\rangle\}.\end{aligned}$$

If the context invokes `modify` with a certain input value v then the store address corresponding to the field `i` is updated. On the other hand, if the context invokes `getPointer`, it returns the `b`'s address, while the environment and the store remain unchanged. Thus a_{b} escapes its scope.

Once an address escapes from an object, the context is free to access the corresponding memory location and to modify it arbitrarily. The transition function for the indirect interactions formalizes this intuition:

DEFINITION 3.15 (INDIRECT INTERACTION, next_{ind}) *Let* $\langle \mathsf{e}, \mathsf{s}, \mathsf{v}, \mathsf{Esc}\rangle \in \Sigma$ *be an interaction state. Then the function* $\text{next}_{\text{ind}} \in [\Sigma \to \mathcal{P}(\Sigma \times \mathsf{Label})]$ *is defined as:*

$$\text{next}_{\text{ind}}(\langle \mathsf{e}, \mathsf{s}, \mathsf{v}, \mathsf{Esc}\rangle) = \{\langle\langle \mathsf{e}, \mathsf{s}', \phi, \mathsf{Esc}\rangle, \kappa\rangle \mid \exists \mathsf{a} \in \mathsf{Esc}.\ \mathsf{s}' \in \text{update}(\mathsf{a}, \mathsf{s})\}.$$

We used the function update in the definition above. It takes as input a pair of memory address a and a store s and it returns the set of the stores where the memory element $\mathsf{s}(\mathsf{a})$ takes all the possible values in the values domain:

DEFINITION 3.16 (update) *The function* update $\in [\mathsf{Addr}\times\mathsf{Store} \to \mathcal{P}(\mathsf{Store})]$ *is defined as:*

$$\text{update}(\mathsf{a},\mathsf{s}) = \{\mathsf{s}' \mid \exists \mathsf{v} \in \mathsf{Val}.\ \mathsf{s}' = \mathsf{s}[\mathsf{a} \mapsto \mathsf{v}]\}.$$

The class constructor does not return any value. Therefore, no address escapes the object scope just after its creation. So, if we refer to the running example of this chapter, it is immediate to see that $\text{next}_{\text{ind}}(S_0\langle\langle 3, \mathsf{true}\rangle, \mathsf{s}\rangle)=\emptyset$. This is a particular case of a more general result, that states that if the object methods do not expose any internal address, then no indirect interaction can happen:

LEMMA 3.1 *Let* $\sigma = \langle \mathsf{e}, \mathsf{s}, \mathsf{v}, \emptyset\rangle$ *be an interaction state. Then* $\text{next}_{\text{ind}}(\sigma) = \emptyset$.

Proof. It follows directly from Definition 3.15: if $\mathsf{Esc} = \emptyset$, then $\text{next}_{\text{ind}}(\sigma)$ is empty.

q.e.d.

EXAMPLE 3.5 (INDIRECT INTERACTION) Let us consider $\langle\sigma_1, \ell\rangle \in \text{next}_{\text{dir}}(\sigma_0)$. If σ_1 is the result of an invocation of the method `modify`, i.e. $\ell = \langle \texttt{modify}, \mathsf{v}\rangle$ for some v, then the lemma above assures that $\text{next}_{\text{ind}}(\sigma_1) = \emptyset$. On the other hand, if $\ell = \langle \texttt{getPointer}, \phi\rangle$ then the context knows the address $\mathsf{a_b}$ and it can arbitrarily access and modify the corresponding store position. Hence:

$$\text{next}_{\text{ind}}(\sigma_1) = \{\langle\langle \mathsf{e}, \mathsf{s}', \phi, \{\mathsf{a_b}\}\rangle, \kappa\rangle \mid \exists \mathsf{v} \in \mathsf{Val}.\ \mathsf{s}' = \mathsf{s}[\mathsf{a_b} \mapsto \mathsf{v}]\}.$$

The transition function for the indirect interaction only takes into account the modification of one memory location at time. However, an indirect interaction that affects several memory addresses can be simulated by the successive applications of the next_{ind} function. This is expressed by the following lemma:

LEMMA 3.2 *Let* $\sigma = \langle \mathsf{e}, \mathsf{s}, \mathsf{v}, \mathsf{Esc}\rangle$ *and* $\sigma' = \langle \mathsf{e}, \mathsf{s}[\mathsf{a_0} \mapsto \mathsf{v_0}, \ldots \mathsf{a_n} \mapsto \mathsf{v_n}], \phi, \mathsf{Esc}\rangle$ *be two interaction states. If* $\forall i.\mathsf{a_i} \in \mathsf{Esc}$ *and* $\forall i, j.\ i \neq j \Rightarrow \mathsf{a_i} \neq \mathsf{a_j}$ *then*

$$\begin{aligned}\exists\sigma_1 \in \Sigma \ldots \exists\sigma_{n-1} \in \Sigma.\ & \langle\sigma_1, \kappa\rangle \in \text{next}_{\text{ind}}(\sigma),\\ & \langle\sigma_2, \kappa\rangle \in \text{next}_{\text{ind}}(\sigma_1), \ldots \langle\sigma', \kappa\rangle \in \text{next}_{\text{ind}}(\sigma_{n-1}).\end{aligned}$$

Proof. The proof is by induction on n:

$n = 0$: The thesis follows immediately by the definition of next_{ind} and update;

$n > 0$: The induction hypothesis implies that there exists a succession of states $\sigma_1 \dots \sigma_{n-1}$ such that $\sigma_{n-1} = \langle \mathsf{e}, \mathsf{s}[\mathsf{a}_0 \mapsto \mathsf{v}_0, \dots \mathsf{a}_{n-1} \mapsto \mathsf{v}_{n-1}], \mathsf{v}, \mathsf{Esc} \rangle$. Then, as all the addresses are distinct, $\text{next}_{\text{ind}}(\sigma_{n-1})$ contains an element $\langle \sigma, \kappa \rangle$.

q.e.d.

It is now possible to define the transition function next, that sums up the direct and indirect interactions:

Definition 3.17 (Transition Function, next) *Let $\sigma \in \Sigma$ be an interaction state. Then the transition function* $\text{next} \in [\Sigma \to \mathcal{P}(\Sigma \times \mathsf{Label})]$ *is defined as:*

$$\text{next}(\sigma) = \text{next}_{\text{dir}}(\sigma) \cup \text{next}_{\text{ind}}(\sigma).$$

Once the transition function is set up, the semantics of an object can be expressed as a least fixpoint on the complete lattice $\langle \mathcal{P}(\mathcal{T}(\Sigma)), \subseteq, \emptyset, \mathcal{T}(\Sigma), \cup, \cap \rangle$. Essentially, the semantics of an object is the set of the traces that encode all the interactions between the object and all the possible contexts it can be instantiated in:

Definition 3.18 (Object Fixpoint Semantics, $\mathfrak{O}[\![\mathsf{o}]\!]$) *Let $v \in \mathsf{Val}$ be an object input value and $s \in \mathsf{Store}$ a store at object creation time. Then the object semantics $\mathfrak{O}[\![\mathsf{o}]\!] \in [\mathsf{D_{in}} \times \mathsf{Store} \to \mathcal{P}(\mathcal{T}(\Sigma))]$ is defined as:*

$$\begin{aligned}\mathfrak{O}[\![\mathsf{o}]\!](\mathsf{v},\mathsf{s}) = \text{lfp}^{\subseteq}_{\emptyset} \lambda T.\ S_0\langle \mathsf{v},\mathsf{s}\rangle \cup \{\sigma_0 \xrightarrow{\ell_0} \dots \xrightarrow{\ell_{n-1}} \sigma_n \xrightarrow{\ell'} \sigma' \mid \\ \sigma_0 \xrightarrow{\ell_0} \dots \xrightarrow{\ell_{n-1}} \sigma_n \in T, \text{next}(\sigma_n) \ni \langle \sigma', \ell' \rangle\}.\end{aligned} \tag{3.1}$$

Using Theorem 2.3 it is immediate to see that

Theorem 3.1 *Let*

$$\begin{aligned}\mathrm{F} = \lambda T.\ S_0\langle \mathsf{v},\mathsf{s}\rangle \cup \{\sigma_0 \xrightarrow{\ell_0} \dots \xrightarrow{\ell_{n-1}} \sigma_n \xrightarrow{\ell'} \sigma' \mid \\ \sigma_0 \xrightarrow{\ell_0} \dots \xrightarrow{\ell_{n-1}} \sigma_n \in T, \text{next}(\sigma_n) \ni \langle \sigma', \ell' \rangle\}.\end{aligned}$$

Then $\mathfrak{O}[\![\mathsf{o}]\!](\mathsf{v},\mathsf{s}) = \bigcup_{n=0}^{\omega} \mathrm{F}^n(\emptyset)$.

Example 3.6 (Object Semantics) We can use the previous theorem to compute (3.1) once it is instantiated with our running example. For the sake of simplicity we do not consider special variables. The first two iterates are:

$$\begin{aligned} I^0 =& \emptyset \\ I^1 =& I^0 \cup \mathrm{F}(\emptyset) = \emptyset \cup S_0\langle\langle 3, \mathsf{true}\rangle, \mathsf{s}\rangle \\ =& \{\langle[\mathtt{i} \mapsto \mathsf{a_i}, \mathtt{b} \mapsto \mathsf{a_b}], \mathsf{s}[\mathsf{a_i} \mapsto 3, \mathsf{a_b} \mapsto \mathsf{true}], \phi, \emptyset\rangle\}. \end{aligned}$$

Let $\mathsf{e_0} = [\mathtt{i} \mapsto \mathsf{a_i}, \mathtt{b} \mapsto \mathsf{a_b}]$, $\mathsf{s_0} = [\mathsf{a_i} \mapsto 3, \mathsf{a_b} \mapsto \mathsf{true}]$ and $\sigma_0 = \langle \mathsf{e_0}, \mathsf{s_0}, \phi, \emptyset\rangle$. Now, from Lemma 3.1 it follows that $\mathrm{next}_{\mathrm{ind}}(\sigma_0) = \emptyset$. Hence, the third iterate of the fixpoint computation is:

$$\begin{aligned} I^2 = I^1 \cup \mathrm{F}(I^1) = \{\sigma_0\} \cup \{\sigma_0 \xrightarrow{\langle \mathtt{modify}, \mathsf{v}\rangle} \langle \mathsf{e_0}, \mathsf{s_0}[\mathsf{a_i} \mapsto 3 + \mathsf{v}], \phi, \emptyset\rangle \mid \mathsf{v} \in \mathsf{Val}\} \\ \cup \{\sigma_0 \xrightarrow{\langle \mathtt{getPointer}, \phi\rangle} \langle \mathsf{e_0}, \mathsf{s_0}, \mathsf{a_b}, \{\mathsf{a_b}\}\rangle\}. \end{aligned}$$

It is worth noting that the invocation of `getPointer` exposes the address $\mathsf{a_b}$ to the context. Therefore, in a successive interaction the context can change the value of the field `b`, and set it to false. This can be seen in the next iteration. Let $\mathsf{s_1}(\mathsf{v}) = \mathsf{s_0}[\mathsf{a_i} \mapsto 3 + \mathsf{v}]$, $\sigma_1' = \langle \mathsf{e_0}, \mathsf{s_1}, \phi, \emptyset\rangle$ and $\sigma_1'' = \langle \mathsf{e_0}, \mathsf{s_0}, \mathsf{a_b}, \{\mathsf{a_b}\}\rangle$,

$$\begin{aligned} I^3 = I^2 \cup \mathrm{F}(I^2) = \{\sigma_0\} \cup \{\sigma_0 \xrightarrow{\langle \mathtt{modify}, \mathsf{v}\rangle} \sigma_1'\} \cup \{\sigma_0 \xrightarrow{\langle \mathtt{getPointer}, \phi\rangle} \sigma_1''\} \\ \cup \{\sigma_0 \xrightarrow{\langle \mathtt{modify}, \mathsf{v}\rangle} \sigma_1' \xrightarrow{\langle \mathtt{modify}, \mathsf{v}_1\rangle} \langle \mathsf{e_0}, \mathsf{s_1}(\mathsf{v} + \mathsf{v}_1), \phi, \emptyset\rangle\} \\ \cup \{\sigma_0 \xrightarrow{\langle \mathtt{modify}, \mathsf{v}\rangle} \sigma_1' \xrightarrow{\langle \mathtt{getPointer}, \phi\rangle} \langle \mathsf{e_0}, \mathsf{s_1}, \mathsf{a_b}, \{\mathsf{a_b}\}\rangle\} \\ \cup \{\sigma_0 \xrightarrow{\langle \mathtt{getPointer}, \phi\rangle} \sigma_1'' \xrightarrow{\langle \mathtt{modify}, \mathsf{v}\rangle} \langle \mathsf{e_0}, \mathsf{s_1}, \phi, \{\mathsf{a_b}\}\rangle\} \\ \cup \{\sigma_0 \xrightarrow{\langle \mathtt{getPointer}, \phi\rangle} \sigma_1'' \xrightarrow{\langle \mathtt{getPointer}, \phi\rangle} \sigma_1''\} \\ \cup \{\sigma_0 \xrightarrow{\langle \mathtt{getPointer}, \phi\rangle} \sigma_1'' \xrightarrow{\kappa} \langle \mathsf{e}, \mathsf{s_0}, \phi, \{\mathsf{a_b}\}\rangle\} \\ \cup \{\sigma_0 \xrightarrow{\langle \mathtt{getPointer}, \phi\rangle} \sigma_1'' \xrightarrow{\kappa} \langle \mathsf{e}, \mathsf{s_0}[\mathsf{a_b} \mapsto \mathsf{false}], \phi, \{\mathsf{a_b}\}\rangle\}. \end{aligned}$$

The iteration schema continues, so we obtain an increasing chain $I^3 \subseteq I^4 \subseteq I^5 \cdots \subseteq I^\omega$. The fixpoint, I^ω, collects all the partial execution traces.

Now we can explain more formally what does it mean that the semantics of an object contains all the possible interactions between an object with any possible context. A generic context is free to interact with the object, by calling any method with any value or by arbitrarily modifying the value of the escaping addresses: the functions $\mathrm{next}_{\mathrm{dir}}$ and $\mathrm{next}_{\mathrm{ind}}$ take into account these cases. As a result, we have that $\mathfrak{O}[\![\mathsf{o}]\!]$ is the most precise property for an object: it describes the behavior of an object during all his lifetime, for each

invocation history and context. Therefore it is an object invariant. However, in general it is not computable so that we need to perform some abstraction in order to obtain an effective object and hence class invariant.

3.3.3 Class Semantics

As a class is a description of a set of objects, it is natural to define the semantics of a class $\mathtt{C}$ as the set that contains the semantics of all the objects that are instances of $\mathtt{C}$. This is expressed by the next definition:

DEFINITION 3.19 (CLASS SEMANTICS, $\mathbb{c}[\![\mathtt{A}]\!]$) *Let* $\mathtt{A} = \langle \mathtt{init}, \mathtt{F}, \mathtt{M} \rangle$. *Then its semantics* $\mathbb{c}[\![\mathtt{A}]\!] \in \mathcal{P}(\mathcal{T}(\Sigma))$ *is*

$$\mathbb{c}[\![\mathtt{A}]\!] = \{\mathbb{o}[\![\mathtt{o}]\!](\mathsf{v},\mathsf{s}) \mid \mathtt{o} \textit{ is an instance of } \mathtt{A}, \mathsf{v} \in \mathsf{D_{in}},\ \mathsf{s} \in \mathsf{Store}\}.$$

A class invariant is a property that is satisfied by the semantics of all the objects that are instance of that class. In our setting, this is equivalent to say that it is satisfied by all the traces in the class semantics. Once again, the most precise class invariant is not computable, so that we need to perform some abstractions in order to obtain an effective analysis.

The class semantics defined above can be expressed in a fixpoint form. Roughly the semantics of a class is the set of semantics of its instances. By definition, the semantics of each instance takes into account, among the others, the interaction with other objects. In particular this is valid when the objects belong to the same class. Therefore, it is possible to "flat" together the semantics of the different instances without introducing new behaviors:

THEOREM 3.2 (CLASS SEMANTICS AS FIXPOINT) *The class semantics can be expressed in fixpoint form*

$$\mathbb{c}[\![\mathtt{A}]\!] = \mathrm{lfp}^{\subseteq}_{\emptyset}\mathrm{F}\langle \mathsf{D_{in}} \times \mathsf{Store}\rangle,$$

where

$$\mathrm{F}\langle S\rangle = \lambda T.\ \{S_0\langle \mathsf{v},\mathsf{s}\rangle \mid \langle \mathsf{v},\mathsf{s}\rangle \in S\} \cup \{\sigma_0 \xrightarrow{\ell_0} \ldots \xrightarrow{\ell_{n-1}} \sigma_n \xrightarrow{\ell'} \sigma' \mid \sigma_0 \xrightarrow{\ell_0} \ldots \xrightarrow{\ell_{n-1}} \sigma_n \in T, \mathrm{next}(\sigma_n) \ni \langle \sigma', \ell'\rangle\}.$$

Proof. By definition $\mathbb{c}[\![\mathtt{A}]\!] = \{\mathbb{o}[\![\mathtt{o}]\!](\mathsf{v},\mathsf{s}) \mid \mathsf{v} \in \mathsf{D_{in}},\ \mathsf{s} \in \mathsf{Store}\}$. Applying in the order Definition 3.18, Theorem 3.1 and Theorem 2.3, we can rewrite the definition of the class semantics as follows:

$$\begin{aligned}\mathbb{c}[\![\mathtt{A}]\!] &= \bigcup_{\langle \mathsf{v},\mathsf{s}\rangle \in \mathsf{D_{in}}\times\mathsf{Store}} \mathrm{lfp}^{\subseteq}_{\emptyset}\mathrm{F}\langle\{\langle \mathsf{v},\mathsf{s}\rangle\}\rangle = \bigcup_{\langle \mathsf{v},\mathsf{s}\rangle \in \mathsf{D_{in}}\times\mathsf{Store}} \bigcup_{n=0}^{\omega} \mathrm{F}^n\langle\{\langle \mathsf{v},\mathsf{s}\rangle\}\rangle(\emptyset)\\ &= \bigcup_{n=0}^{\omega} \mathrm{F}^n\langle \mathsf{D_{in}} \times \mathsf{Store}\rangle(\emptyset) = \mathrm{lfp}^{\subseteq}_{\emptyset}\mathrm{F}\langle \mathsf{D_{in}} \times \mathsf{Store}\rangle.\end{aligned}$$

q.e.d.

3.4 Relation between $\mathbb{w}[\![\cdot]\!]$ and $\mathbb{c}[\![\cdot]\!]$

We are now left to study the relation between $\mathbb{w}[\![\mathtt{P}]\!]$, the semantics of a whole object-oriented program, and $\mathbb{c}[\![\mathtt{A}]\!]$, the semantics of a class. The main result of this section are Theorems 3.3 and 3.4, which state the soundness and the completeness of the class semantics w.r.t. the whole program semantics.

3.4.1 Abstraction

We define an abstraction function that, given a trace $\tau \in \mathcal{T}(\mathsf{Env} \times \mathsf{Store})$ and an object o that belongs to a class $\mathtt{A}$, *cuts* all the sub-traces of τ that do *not* involve the object o.

First, we define the helper function $\text{split}^{\circ} \in [\mathcal{T}(\mathsf{Env} \times \mathsf{Store}) \to (\mathsf{Env} \times \mathsf{Store}) \times \mathcal{T}(\mathsf{Env} \times \mathsf{Store})]$ which, given a trace τ and an object o, finds the longest prefix of τ made up of states involving the execution of a method of o and returns a pair consisting of the last state of such a prefix and the remaining suffix of τ. Formally:

DEFINITION 3.20 (split°) *Let* o *be an object and* $\tau \in \mathcal{T}(\mathsf{Env} \times \mathsf{Store})$ *and* $\mathtt{pc}_{\mathtt{exit}}$ *be the exit point of* $\tau(0)(\mathtt{curMethod})$. *Then* $\text{split}^{\circ} \in [\mathcal{T}(\mathsf{Env} \times \mathsf{Store}) \to (\mathsf{Env} \times \mathsf{Store}) \times \mathcal{T}(\mathsf{Env} \times \mathsf{Store})]$ *is defined as*

$$\begin{aligned}\text{split}^{\circ}(\tau) = &\mathit{let}\ n = \min\{i \in \mathbb{N} \mid \tau(i)(\mathtt{pc}) = \mathtt{pc}_{\mathtt{exit}} \ \wedge \tau(i)(\mathtt{this}) = \mathsf{o} \ \wedge \\ &\qquad \tau(0)(\mathtt{stackHeight}) = \tau(i)(\mathtt{stackHeight})\} \\ &\mathit{in}\ \langle \tau(n), \tau(n+1) \to \ldots \to \tau(\text{len}(\tau) - 1)\rangle.\end{aligned}$$

DEFINITION 3.21 ($\alpha^{\circ}_{\preceq}$) *Let* o *be an object,* $\tau \in \mathcal{T}(\mathsf{Env} \times \mathsf{Store})$. *Then* $\alpha^{\circ}_{\preceq} \in [(\mathcal{T}(\mathsf{Env} \times \mathsf{Store}) \times \mathsf{Store}) \to \mathcal{T}(\mathsf{Env} \times \mathsf{Store})]$ *is defined as:*

$$\alpha^{\circ}_{\preceq} = \lambda(\tau, \mathsf{s}_{\mathsf{last}}).\ \begin{cases} \epsilon & \mathit{if}\ \tau = \epsilon \\ \mathit{let}\ \langle \rho', \tau'\rangle = \text{split}^{\circ}(\tau) \\ \mathit{in}\ \ \mathit{let}\ \langle \mathsf{e}', \mathsf{s}'\rangle = \rho' & \mathit{if}\ \tau = \langle \mathsf{e}, \mathsf{s}\rangle \to \tau'', \mathsf{e}(\mathtt{this}) = \mathsf{o} \\ \mathit{in}\ \ \rho' \to \alpha^{\circ}_{\preceq}(\tau', \mathsf{s}') & \\ \alpha^{\circ}_{\preceq}(\tau'', \mathsf{s}_{\mathsf{last}}) & \mathit{if}\ \tau = \langle \mathsf{e}, \mathsf{s}\rangle \to \tau'', \mathsf{e}(\mathtt{this}) \neq \mathsf{o}, \\ & \mathsf{s}_{/\mathsf{s}(\mathsf{o})} = \mathsf{s}_{\mathsf{last}/\mathsf{s}(\mathsf{o})} \\ \langle \mathsf{e}, \mathsf{s}\rangle \to \alpha^{\circ}_{\preceq}(\tau'', \mathsf{s}) & \mathit{if}\ \tau = \langle \mathsf{e}, \mathsf{s}\rangle \to \tau'', \mathsf{e}(\mathtt{this}) \neq \mathsf{o}, \\ & \mathsf{s}_{/\mathsf{s}(\mathsf{o})} \neq \mathsf{s}_{\mathsf{last}/\mathsf{s}(\mathsf{o})}. \end{cases}$$

The four cases of $\alpha^{\circ}_{\asymp}$ can be explained as follows. The abstraction of the empty trace is the empty trace. If the first state of the trace involves the object o (i.e. $\mathsf{e}(\mathtt{this}) = \mathsf{o}$), then it means we are inside some method $\mathtt{m}$ of o, so that we abstract away all the internal steps of the execution of $\mathtt{m}$ but the last and we apply recursively the abstraction to the rest of the trace. If the current object is not o, then we can distinguish two cases for the first state of τ. If the part of memory reached by the environment of o is *not* changed (i.e. $\mathsf{s}_{/\mathsf{s}(\mathsf{o})} = \mathsf{s}_{\mathsf{last}/\mathsf{s}(\mathsf{o})}$) then it can be dropped. If not, it must be kept. In the two cases we continue with the rest of the trace (i.e. τ'').

The next abstraction maps the states of a trace to interaction states.

DEFINITION 3.22 ($\alpha^{\circ}_{\uparrow}$) *Let* o *be an object and* $\tau \in \mathcal{T}(\mathsf{Env} \times \mathsf{Store})$. *Then* $\alpha^{\circ}_{\uparrow} \in [(\mathcal{T}(\mathsf{Env} \times \mathsf{Store}) \times \mathcal{P}(\mathsf{Addr})) \to \mathcal{T}(\Sigma)]$ *is defined as*

$$\alpha^{\circ}_{\uparrow} = \lambda(\tau, \mathsf{Esc}).\ \begin{cases} \epsilon & \text{if } \tau = \epsilon \\ \begin{array}{ll} \textit{let} & \langle \mathsf{e}, \mathsf{s}\rangle = \rho \\ \textit{in} & \textit{let } \mathsf{Esc}' = \mathsf{Esc} \cup \mathrm{reachable}(\rho(\mathtt{retVal}), \mathsf{s}) \\ \textit{in} & \langle\langle \mathsf{e}, \mathsf{s}, \rho(\mathtt{retVal}), \mathsf{Esc}'\rangle, \\ & \ \langle \rho(\mathtt{curMethod}), \rho(\mathtt{inVal})\rangle\rangle \\ & \to \alpha^{\circ}_{\uparrow}(\tau', \mathsf{Esc}') \end{array} & \begin{array}{l} \text{if } \tau = \rho \to \tau', \\ \mathsf{e}(\mathtt{this}) = \mathsf{o} \end{array} \\ \begin{array}{ll} \textit{let} & \langle \mathsf{e}, \mathsf{s}\rangle = \rho \\ \textit{in} & \langle\langle \mathsf{e}, \mathsf{s}, \phi, \mathsf{Esc}\rangle, \kappa\rangle \to \alpha^{\circ}_{\uparrow}(\tau', \mathsf{Esc}) \end{array} & \begin{array}{l} \text{if } \tau = \rho \to \tau', \\ \mathsf{e}(\mathtt{this}) \neq \mathsf{o}. \end{array} \end{cases}$$

Finally, the abstraction $\alpha^{\circ} \in [\mathcal{P}(\mathcal{T}(\mathsf{Env} \times \mathsf{Store})) \to \mathcal{P}(\mathcal{T}(\Sigma))]$, given a set of execution states T, projects from the traces in T the states that *are relevant* to the object o, either by calling one of its methods or by modifying the value of a memory location reachable by the fields of o. Formally:

DEFINITION 3.23 (α°) *Let* o *be an object,* $T \subseteq \mathcal{T}(\mathsf{Env} \times \mathsf{Store})$ *a set of execution traces and* $\mathsf{s}_{\emptyset}$ *be the empty store. Then the abstraction* $\alpha^{\circ} \in [\mathcal{P}(\mathcal{T}(\mathsf{Env} \times \mathsf{Store})) \to \mathcal{P}(\mathcal{T}(\Sigma))]$ *is defined as*

$$\alpha^{\circ}(T) = \{\alpha^{\circ}_{\uparrow}(\alpha^{\circ}_{\asymp}(\tau, \mathsf{s}_{\emptyset}), \emptyset) \mid \tau \in T\}.$$

LEMMA 3.3 (α° IS A COMPLETE $\cup$-MORPHISM) $\forall\{T_i \mid T_i \subseteq \mathcal{T}(\mathsf{Env}\times\mathsf{Store})\}$. $\alpha^{\circ}(\bigcup_i T_i) = \bigcup_i \alpha^{\circ}(T_i)$.

Proof. Let $\{T_i\}$ a collection of set of traces. Then

$$\begin{aligned} \alpha^{\circ}(\bigcup_i T_i) &= \{\alpha^{\circ}_{\uparrow}(\alpha^{\circ}_{\asymp}(\tau, \mathsf{s}_{\emptyset}), \emptyset) \mid \tau \in \bigcup_i T_i\} \\ &= \bigcup_i \{\alpha^{\circ}_{\uparrow}(\alpha^{\circ}_{\asymp}(\tau, \mathsf{s}_{\emptyset}), \emptyset) \mid \tau \in T_i\} = \bigcup_i \alpha^{\circ}(T_i). \end{aligned}$$

q.e.d.

The previous lemma, together with Lemma 2.4 and Lemma 2.3 imply that there exists a concretization function γ° such that:

LEMMA 3.4 (SOUNDNESS OF α°)

$$\langle \mathcal{P}(\mathcal{T}(\mathsf{Env} \times \mathsf{Store})), \subseteq, \emptyset, \mathcal{T}(\mathsf{Env} \times \mathsf{Store}), \cup, \cap \rangle \underset{\alpha^\circ}{\overset{\gamma^\circ}{\leftrightarrows}} \langle \mathcal{P}(\mathcal{T}(\Sigma)), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle.$$

3.4.2 Soundness and Completeness of the Class Semantics

We can now state the relation between the whole-program semantics $\mathbb{w}[\![\cdot]\!]$ and the class semantics $\mathbb{c}[\![\cdot]\!]$. The first theorem essentially states that all the interactions that a program `P` performs on an object `o` instance of a class `A` are captured by $\mathbb{c}[\![\texttt{A}]\!]$.

THEOREM 3.3 (SOUNDNESS OF $\mathbb{c}[\![\cdot]\!]$) *Let* `P` *be a whole object-oriented program, let* `A` $\in$ `P` *and* `o` *be an instance of* `A`. *Then*

$$\forall R_0 \subseteq \mathsf{Env} \times \mathsf{Store}. \forall \tau \in \mathcal{T}(\mathsf{Env} \times \mathsf{Store}). \\ \tau \in \mathbb{w}[\![\texttt{P}]\!](R_0) \Longrightarrow \exists \tau' \in \mathbb{c}[\![\texttt{A}]\!].\ \alpha^\circ(\{\tau\}) = \{\tau'\}.$$

Proof. Let $\tau \in \mathbb{w}[\![\texttt{P}]\!](R_0)$. By definition of $\mathbb{w}[\![\cdot]\!]$, $\tau = \rho_0 \rightarrow \ldots \rightarrow \rho_n$, with $\rho_o \in R_0$. If `o` is not an object instantiated in τ, then $\forall i \in [0..n].\ \tau(i)(\texttt{this}) \neq \mathsf{o}$, so that by definition of α°, $\alpha^\circ(\{\tau\}) = \{\epsilon\}$, so that the theorem is straightforwardly verified. If not, $\alpha^\circ(\{\tau\}) = \{\tau_\alpha\}$ is such that $\tau_\alpha = \sigma_0 \xrightarrow{\ell_0} \ldots \xrightarrow{\ell_{k-1}} \sigma_k$, with $k \leq n$ and $\forall i \in [0..k].\ \sigma_i \in \Sigma$ and either $\ell_i = \kappa$ or $\ell_i = \langle \mathtt{m}, \mathsf{v} \rangle$. The first state, σ_0 is the consequence of the object creation, i.e. of the constructor invocation. Therefore, as $\mathbb{i}[\![\texttt{init}]\!]$ is built on the top of $\mathbb{w}_\mathbb{m}[\![\texttt{init}]\!]$, Definition 3.7 and Definition 3.11 imply that $\exists i \in [0..n].\ \rho_i = \langle \mathsf{e}_i, \mathsf{s}_i \rangle . \exists \mathtt{x} \in \texttt{Vars}.\ \sigma_0 \in S_0\langle \rho_i(\mathtt{x}), \mathsf{s}_i \rangle$. Now, consider for all the $i \geq 0$ the sub-trace $\sigma_i \xrightarrow{\ell_i} \sigma_{i+1}$ of τ_α. If $\ell_i = \kappa$ then by definition of next_{ind}, $\sigma_{i+1} \in \text{next}_{\text{ind}}(\sigma_i)$. Otherwise, $\ell = \langle \mathtt{m}, \mathsf{v} \rangle$ for some method `m` of `A` and `v` $\in \mathsf{D}_{\mathsf{in}}$. Therefore, as next_{dir} considers all the methods and all the possible input value, $\sigma_{i+1} \in \text{next}_{\text{ind}}(\sigma_i)$. To sum up, the fact that $\sigma_0 \in S_0\langle \rho_i(\mathtt{x}), \mathsf{s}_i \rangle$ and that all the transitions of τ_α may be mimicked by either next_{ind} or next_{dir} implies that $\tau_\alpha \in \mathbb{c}[\![\texttt{A}]\!]$ necessarily.

q.e.d.

Next theorem states that all the behaviors considered by $\mathbb{c}[\![\texttt{A}]\!]$ are feasible.

THEOREM 3.4 (COMPLETENESS OF $\mathbb{c}[\![\cdot]\!]$) *Let* $\mathtt{A}$ *be a class. Then*

$$\forall \tau \in \mathcal{T}(\Sigma).\ \tau \in \mathbb{c}[\![\mathtt{A}]\!] \Longrightarrow \exists \mathtt{P}.\exists \rho_o \in \mathsf{Env} \times \mathsf{Store}.\ \exists \mathtt{o}\ \textit{instance of}\ \mathtt{A}.\exists \tau' \in \mathcal{T}(\mathsf{Env} \times \mathsf{Store}).\ \tau' \in \mathbb{w}[\![\mathtt{P}]\!](\{\rho_0\}) \wedge \alpha^{\circ}(\{\tau'\}) = \{\tau\}.$$

Proof. (Sketch) Let $\tau = \sigma_0 \xrightarrow{\ell_0} \ldots \xrightarrow{\ell_{n-1}} \sigma_n \in \mathbb{c}[\![\mathtt{A}]\!]$. The theorem is proved by constructing a suitable program $\mathtt{P}$ and initial state ρ_0. By definition of $\mathbb{c}[\![\mathtt{A}]\!]$ and Definition 3.11, it exists a pair $\langle \mathsf{v}, \mathsf{s} \rangle$ such that $\mathbb{i}[\![\mathtt{init}]\!]\langle \mathsf{v}, \mathsf{s} \rangle = \sigma_0$. As a consequence, we can build a $\rho_0 = \langle \mathsf{e}', \mathsf{s} \rangle$ such that e' is an environment that satisfy $\mathsf{s}(\mathsf{e}'(\mathtt{inVal})) = \mathsf{v}$. The corresponding program $\mathtt{P}$ will contain the line $\mathtt{o} = \mathtt{new}\ \mathtt{A}(\mathtt{inVal})$. Then, the construction of the other lines of $\mathtt{P}$ goes on by considering the labels $\ell_0, \ell_1, \ldots \ell_{n-1}$: $\forall i \in [0..n-1]$ if $\ell_i = \langle \mathtt{m}, \mathsf{v} \rangle$ then $\mathtt{P}$ will contain the command $\mathtt{o.m}(\mathsf{v})$. If not, for a transition $\sigma_i \xrightarrow{\kappa} \sigma_{i+1}$ it will contain a sequence of assignments $*\mathsf{a_j} = \mathsf{v}$ where $\mathsf{a_j}$ is the address of the an escaped field value (obtained from the field $\mathtt{Esc}$ of σ_i) and v is the new value (obtained from the store in σ_{i+1}, i.e. $\mathsf{s}_{i+1}(\mathsf{a_j}) = \mathsf{v}$). By construction of ρ_0 and $\mathtt{P}$ the theorem is proved.

q.e.d.

3.5 Languages with Class Destructor

Most class-based object oriented languages as C++ [101] or Object Pascal [12] provide a class destructor, somehow a dual of the class constructor. In simple terms a destructor is a member that implements the actions required to destroy an instance of a class. For instance, destructors can be used to recover the heap space or to terminate file I/O that is associated with the removed class instance. In this section we sketch how the previous definitions and results can be extended to cope with languages that provide a class destructor.

First, the class syntax can be extended with a further element: the class destructor `destroy`:

$$\mathtt{D} = \langle \mathtt{init}, \mathtt{F}, \mathtt{M}, \mathtt{destroy} \rangle.$$

The semantics of an object should take into account the destructor so that all the traces must end with a state consequence of the invocation of the destructor. As a consequence, all the maximal traces begin with an "init" transition and they end with a "destroy" transition. Such a semantics can be obtained from the trace semantics above by filtering the suitable traces. In particular,

we keep all the traces that end with an invocation of the destructor:

$$\mathbb{c}'[\![\mathtt{D}]\!] = \{\sigma_0 \xrightarrow{\ell_0} \ldots \xrightarrow{\texttt{destroy}} \sigma_n \in \mathbb{c}[\![\mathtt{D}]\!]\}.$$

Nevertheless, in this thesis we do not consider classes endowed with a class destructors. The main reason is that modern object oriented languages come with a garbage collector, so that the use of a destructor to explicitly deallocate an object is not required, if not forbidden. For example in Java [56] the memory management is left to the JVM, so that as soon as an object is no more *accessible* then it becomes a candidate for removing. However, it still has a form of destructor, namely the `finalize` method, which is invoked by the garbage collector just before it frees the memory allocated by the object. The main difference between a C++ destructor and the `finalize` method is that the first can be explicitly invoked by the program text whereas this is denied for the latter.

Chapter 4

Abstract Semantics

Per aspera ad astra. [1]

Latin proverb

The class concrete semantics $\mathbb{c}[\![\texttt{A}]\!]$ is the most precise class property. Unfortunately, it is non-computable: in the general case, $\mathbb{c}[\![\texttt{A}]\!]$ is a set that contains infinitely many traces. Therefore, we need to perform some abstractions in order to obtain a computable approximation of the class semantics. In this chapter we present a generic, two-steps abstraction for inferring the reachable states of a class. The first abstraction collects together all the states that have the same cause, e.g. they are the result of the invocation of the same methods. The nodes of the traces will be no more states but sets of states, and for each node there will be one, and just one, successor for each class method. Next step abstracts away the computational history. In other terms, the abstraction collects all the nodes in the traces, forgetting the casual relation between the states. Finally, the last step will be to abstract the set of objects that may be instances of a class.

4.1 Stepwise Abstraction

In the *stepwise methodology* to the design of abstract semantics, the concrete semantics is approximated by a series of successive abstractions, the intuition being that each step abstracts away the properties that are not relevant for the analysis. Examples of stepwise abstraction can be found in [28] to obtain a hierarchy of semantics as stepwise abstractions of a concrete trace semantics and in [26] to derive a hierarchy of type systems systematically from

[1] (Latin) Through difficult things to the abstract.

a denotational semantics. As far as static analysis is concerned, examples are [27] and [45] where the approach has been used for the design of generic abstract interpreters respectively for a small imperative language and for the π-calculus.

The stepwise approach to the design of abstract semantics is theoretically justified by Theorem 2.5, which states that the set of abstractions of a given concrete domain forms a complete lattice. Such a methodology can be sketched as follows. Let us have a concrete domain $\langle \mathsf{D}, \sqsubseteq \rangle$, which is a complete lattice, and a semantic function $\mathbb{s}[\![\mathtt{A}]\!]$ defined on D. The first step is to define the abstract domain, say $\bar{\mathsf{D}}$, and the abstraction function $\alpha \in [\mathsf{D} \to \bar{\mathsf{D}}]$ that relates the elements of D and $\bar{\mathsf{D}}$. Intuitively, the abstract domain $\bar{\mathsf{D}}$ captures the properties of the semantics we are interested in. Next, one must show the soundness of the abstraction, e.g. that the concrete and the abstract domain are related by a Galois connection. For instance, this can be done either by explicitly giving the concretization map γ or by proving that α is a complete join morphism (cf. Lemma 2.3). Finally, the abstract semantics $\bar{\mathbb{s}}[\![\mathtt{A}]\!]$ is designed by calculus as an upper approximation of the concrete semantics:

$$\alpha \circ \mathbb{s}[\![\mathtt{A}]\!] \bar{\sqsubseteq} \bar{\mathbb{s}}[\![\mathtt{A}]\!]. \tag{4.1}$$

If the so-obtained abstract semantics is still not satisfactory, then the process can be repeated starting from the abstract domain $\bar{\mathsf{D}}$ and the corresponding abstract semantics. As a consequence, the stepwise approach produces a chain of abstract domains (cf. Theorem 2.5):

$$\langle \mathsf{D}, \sqsubseteq \rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \bar{\mathsf{D}}_1, \bar{\sqsubseteq}_1 \rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \ldots \underset{\alpha_n}{\overset{\gamma_n}{\leftrightarrows}} \langle \bar{\mathsf{D}}_n, \bar{\sqsubseteq}_n \rangle \tag{4.2}$$

and of respective sound abstract semantics $\bar{\mathbb{s}}_i[\![\mathtt{A}]\!]$. Roughly, each element in the chain (4.2) represents an abstraction from a particular property. Therefore the refinement of the abstract semantics w.r.t. a particular property can be easily obtained by refining the corresponding abstraction.

Example 4.1 (Temporal Class Invariants) Later in this chapter we introduce a state-based abstraction of the class semantics that abstracts away the casual relations between states. Nevertheless, an abstract semantics that describes the objects evolution in time [41] can be obtained replacing the state-based abstraction with e.g. Typed Decision Graphs [79].

From Theorem 2.5 it follows that $\langle \bar{\mathsf{D}}_n, \bar{\sqsubseteq}_n \rangle$ is a sound abstraction of the concrete domain $\langle \mathsf{D}, \sqsubseteq \rangle$ and from Lemma 2.1 and Theorem 2.4 it follows that the corresponding abstract semantics $\bar{\mathbb{s}}_n[\![\mathtt{A}]\!]$ is an upper approximation of $\mathbb{s}[\![\mathtt{A}]\!]$:

$$\alpha_n \circ \alpha_{n-1} \circ \ldots \alpha_1 \circ \mathbb{s}[\![\mathtt{A}]\!] \bar{\sqsubseteq}_n \bar{\mathbb{s}}_n[\![\mathtt{A}]\!]. \tag{4.3}$$

$$\begin{array}{l}\sigma_1 \xrightarrow{\langle \mathtt{m_i}, \mathtt{v_1} \rangle} \sigma_2 \xrightarrow{\langle \mathtt{m_j}, \mathtt{v_2} \rangle} \sigma_3 \\ \sigma'_1 \xrightarrow{\langle \mathtt{m_i}, \mathtt{v'_1} \rangle} \sigma'_2 \xrightarrow{\langle \mathtt{m_j}, \mathtt{v'_2} \rangle} \sigma'_3 \\ \sigma''_1 \xrightarrow{\langle \mathtt{m_i}, \mathtt{v''_1} \rangle} \sigma''_2 \xrightarrow{\kappa} \sigma''_3\end{array} \quad \overset{\alpha_{\succ}}{\Longrightarrow} \quad \begin{array}{l}\{\sigma_1, \sigma'_1, \sigma''_1\} \xrightarrow{\mathtt{m_i}} \{\sigma_2, \sigma'_2, \sigma''_2\} \xrightarrow{\mathtt{m_j}} \{\sigma_3, \sigma'_3\} \\ \{\sigma_1, \sigma'_1, \sigma''_1\} \xrightarrow{\mathtt{m_i}} \{\sigma_2, \sigma'_2, \sigma''_2\} \xrightarrow{\kappa} \{\sigma''_3\}\end{array}$$

Figure 4.1: The first abstraction

The soundness proof of $\bar{\mathbb{s}}_n[\![\mathtt{A}]\!]$ boils down in proving that at each abstraction step the abstract domain and the abstract semantics verify (4.1). In general such a proof is easier than a direct approach, that can be longer, more complex and error-prone. Furthermore, the soundness is ensured by construction, i.e. the abstract semantic function is derived by calculus.

To sum up, the main advantages of a stepwise-abstraction design of abstract semantics are that:

- a lengthy and error-prone soundness prove is split in smaller, easier-to-check proofs;
- the soundness of the abstract semantics is by construction;
- the points where precision is lost are clearly identified;
- the refinement of the analysis boils down to the refinement of the corresponding abstraction.

A further advantage will be made clearer in the next chapter. In fact, thanks to the stepwise approach, the soundness of a class invariant is proved once and for all. Thus, an instance of the framework must just prove the soundness w.r.t. the semantics of the method bodies.

4.2 First Abstraction: Collecting Traces

The semantics of an object is a set of labeled traces. Each node in one of these traces is an interaction state and the trace labels encode the interaction history of the object. The aim of the first abstraction is to "smash" together all the traces that share the same interaction history. For example, let us consider the traces in Figure 4.1: the first two traces are consequence of the sequential invocation of the methods $\mathtt{m_i}$ and $\mathtt{m_j}$ whereas the third one has a different interaction history. The abstraction function $\alpha_{\succ}$ collects together all the traces and trace prefixes that have the same labels, i.e. the same invocation history. It abstracts away the method input values $\mathtt{v}$'s and the

causality relation between the states. For example in Figure 4.1, the fact the state σ_3'' is the successor of σ_2'' is lost: it can be either a successor of σ_2, σ_2' or σ_2''. Nevertheless the property that it is consequence of an indirect interaction is preserved.

In the rest of this section we formally define the abstract domain of the collecting traces, the abstraction and concretization functions and eventually we derive the abstract semantics by calculus.

4.2.1 Abstract Domain

We consider an abstract domain whose elements are sets of traces that we call *collecting traces*. A node in a collecting trace is a set of states and each transition is labeled with the cause that originated it. Furthermore we require that each element in the abstract domain contains only one collecting trace with a given interaction history. Differently stated, we forbid that two traces with the same interaction history coexist in the same set. The order, the smallest and the largest element, the join and the meet are defined accordingly. We start by formally defining the interaction history:

Definition 4.1 (Interaction History) *The interaction history is given by the function* $\text{history} \in [\mathcal{T}(\Sigma) \to \{\mathtt{M} \cup \{\kappa\}\}^*]$ *defined as follows:*

$$\begin{aligned}
\text{history}(\epsilon) &= \epsilon \\
\text{history}(\sigma \xrightarrow{\langle \mathtt{m},\mathtt{v}\rangle} \tau) &= \mathtt{m} \cdot \text{history}(\tau) \\
\text{history}(\sigma \xrightarrow{\mathtt{m}} \tau) &= \mathtt{m} \cdot \text{history}(\tau) \\
\text{history}(\sigma \xrightarrow{\kappa} \tau) &= \kappa \cdot \text{history}(\tau).
\end{aligned}$$

Example 4.2 With reference to the traces in Figure 4.1, we have that:

$$\text{history}\left(\sigma_1 \xrightarrow{\langle \mathtt{m_i},\mathtt{v_1}\rangle} \sigma_2 \xrightarrow{\langle \mathtt{m_j},\mathtt{v_2}\rangle} \sigma_3\right) = \text{history}\left(\sigma_1' \xrightarrow{\langle \mathtt{m_i},\mathtt{v_1'}\rangle} \sigma_2' \xrightarrow{\langle \mathtt{m_j},\mathtt{v_2'}\rangle} \sigma_3'\right) = \mathtt{m_i} \cdot \mathtt{m_j}$$

$$\text{history}\left(\sigma_1'' \xrightarrow{\langle \mathtt{m_i},\mathtt{v_1''}\rangle} \sigma_2'' \xrightarrow{\kappa} \sigma_3''\right) = \mathtt{m_i} \cdot \kappa$$

It is worth noting that the interaction history abstracts away from the method input value v.

The abstract elements are sets of collecting traces that do not contain *doubles* w.r.t. the interaction history. Namely, if an element contains two traces with the same interaction history then they are the same trace:

Definition 4.2 (Sets of Collecting Traces) *The set of collecting traces* $\bar{\mathsf{D}}_{\succ} \subseteq \mathcal{P}(\mathcal{T}(\mathcal{P}(\Sigma)))$ *is such that*

$$\forall \bar{\mathsf{T}} \in \bar{\mathsf{D}}_{\succ}.\ \forall \bar{\mathsf{t}}_1, \bar{\mathsf{t}}_2 \in \bar{\mathsf{T}}.\ \text{history}(\bar{\mathsf{t}}_1) = \text{history}(\bar{\mathsf{t}}_2) \Rightarrow \bar{\mathsf{t}}_1 = \bar{\mathsf{t}}_2.$$

Given two sets of collecting traces $\bar{\mathsf{T}}_1$ and $\bar{\mathsf{T}}_2$, $\bar{\mathsf{T}}_1$ is larger than $\bar{\mathsf{T}}_2$ if it is able to *mimic* its the behavior. By mimic we mean that for each collecting trace in $\bar{\mathsf{T}}_2$ there exists one in $\bar{\mathsf{T}}_1$ that has the same history trace and *involves* at least the same states:

DEFINITION 4.3 (ORDER, $\bar{\subseteq}_\succ$) *Let $\bar{\mathsf{T}}_1$ and $\bar{\mathsf{T}}_2$ be two elements of $\bar{\mathsf{D}}_\succ$. Then*

$$\bar{\mathsf{T}}_1 \bar{\subseteq}_\succ \bar{\mathsf{T}}_2 \iff \forall \bar{\mathsf{t}}_1 \in \bar{\mathsf{T}}_1. \exists \bar{\mathsf{t}}_2 \in \bar{\mathsf{T}}_2.\ \bar{\mathsf{t}}_1 \bar{\subseteq}^\tau_\succ \bar{\mathsf{t}}_2,$$

where the order $\bar{\subseteq}^\tau_\succ$ on single traces is defined as

$$\begin{aligned}\bar{\mathsf{t}}_1 \bar{\subseteq}^\tau_\succ \bar{\mathsf{t}}_2 \Longleftrightarrow & (\bar{\mathsf{t}}_1 = \bar{\mathsf{t}}_2) \vee \\ & (\bar{\mathsf{t}}_1 = S_1 \xrightarrow{\ell_1} \bar{\mathsf{t}}'_1 \wedge \bar{\mathsf{t}}_2 = S_2 \xrightarrow{\ell_2} \bar{\mathsf{t}}'_2 \wedge S_1 \subseteq S_2 \wedge \ell_1 = \ell_2 \wedge \bar{\mathsf{t}}'_1 \bar{\subseteq}^\tau_\succ \bar{\mathsf{t}}'_2)\end{aligned}$$

The definition of $\bar{\subseteq}^\tau_\succ$ above can be rewritten in order to make explicit the fact that the two traces have the same interaction history [1]:

$$\begin{aligned}\bar{\mathsf{t}}_1 \bar{\subseteq}^\tau_\succ \bar{\mathsf{t}}_2 \Longleftrightarrow & \mathrm{history}(\bar{\mathsf{t}}_1) = \mathrm{history}(\bar{\mathsf{t}}_2) \wedge \\ & (\forall i.\ \bar{\mathsf{t}}_1(i) = S_i \wedge \bar{\mathsf{t}}_2(i) = S'_i \Rightarrow S_i \subseteq S'_i).\end{aligned}$$

EXAMPLE 4.3 Let us consider the following collecting traces:

$$\begin{aligned}\bar{\mathsf{t}}_1 = &\ \{\sigma_1, \sigma_2\} \xrightarrow{\ell_1} \{\sigma'_1, \sigma'_2\} \xrightarrow{\ell_2} \{\sigma''_1, \sigma''_2, \sigma''_3\} \\ \bar{\mathsf{t}}_2 = &\ \{\sigma_1, \sigma_2\} \xrightarrow{\ell_1} \{\sigma'_1, \sigma'_2, \sigma'_3\} \xrightarrow{\ell_2} \{\sigma''_1, \sigma''_2, \sigma''_3\} \\ \bar{\mathsf{t}}_3 = &\ \{\sigma_1, \sigma_2, \sigma_3\} \xrightarrow{\ell_1} \{\sigma'_4\} \xrightarrow{\ell_2} \{\sigma''_1, \sigma''_2, \sigma''_3\},\end{aligned}$$

applying the definition above, it is easy to see that $\bar{\mathsf{t}}_1 \bar{\subseteq}^\tau_\succ \bar{\mathsf{t}}_2$. However, neither $\bar{\mathsf{t}}_2 \bar{\subseteq}^\tau_\succ \bar{\mathsf{t}}_3$ nor $\bar{\mathsf{t}}_3 \bar{\subseteq}^\tau_\succ \bar{\mathsf{t}}_2$.

Once we have the order it is quite easy to give the smallest and the largest elements w.r.t. this order. It is routine to check that $\bar{\bot}_\succ = \emptyset$ is the smallest element. On the other hand, the largest element in the abstract domain must contain all the possible interaction histories. Furthermore, each node in a trace must contain all the possible states. Therefore it is immediate to verify that:

LEMMA 4.1 (TOP, $\bar{\top}_\succ$) *The set of collecting traces $\bar{\top}_\succ \in \bar{\mathsf{D}}_\succ$, defined as*

$$\bar{\top}_\succ = \bigcup_{h \in \{\mathtt{M} \cup \{\kappa\}\}^*} \{\bar{\mathsf{t}} \mid \forall i. \bar{\mathsf{t}}(i) = \Sigma, \mathrm{history}(\bar{\mathsf{t}}) = h\}.$$

is such that $\forall \bar{\mathsf{T}} \in \bar{\mathsf{D}}_\succ. \bar{\mathsf{T}} \bar{\subseteq}_\succ \bar{\top}_\succ$.

[1] We recall that $\bar{\mathsf{t}}(i)$ denotes the i-th node of the trace $\bar{\mathsf{t}}$.

The join operator collects together the states of the traces that have the same interaction history:

DEFINITION 4.4 (JOIN, $\bar{\cup}_\succ$) *Let* $\bar{\mathsf{T}}_1$ *and* $\bar{\mathsf{T}}_2$ *be two elements of* $\bar{\mathsf{D}}_\succ$*, then the join operator* $\bar{\cup}_\succ \in [\bar{\mathsf{D}}_\succ \times \bar{\mathsf{D}}_\succ \to \bar{\mathsf{D}}_\succ]$ *is*

$$\bar{\mathsf{T}}_1 \bar{\cup}_\succ \bar{\mathsf{T}}_2 = \{\bar{\cup}^\tau_\succ \bar{\mathsf{T}} \mid \bar{\mathsf{T}} \in (\bar{\mathsf{T}}_1 \cup \bar{\mathsf{T}}_2)_{/\text{history}}\},$$

where $\bar{\cup}^\tau_\succ \in [\mathcal{T}(\mathcal{P}(\Sigma)) \times \mathcal{T}(\mathcal{P}(\Sigma)) \to \mathcal{T}(\mathcal{P}(\Sigma))]$ *is defined as follows:*

$$S_1 \bar{\cup}_\succ S_2 = S_1 \cup S_2$$
$$S_1 \xrightarrow{\ell} \tau_1 \bar{\cup}^\tau_\succ S_2 \xrightarrow{\ell} \tau_2 = S_1 \cup S_2 \xrightarrow{\ell} (\tau_1 \bar{\cup}^\tau_\succ \tau_2).$$

EXAMPLE 4.4 If we apply the definition above to the traces of the Example 4.3, we obtain:

$$\{\bar{\mathsf{t}}_2\} \bar{\cup}_\succ \{\bar{\mathsf{t}}_3\} = \underbrace{\{\{\sigma_1, \sigma_2, \sigma_3\} \xrightarrow{\ell_1} \{\sigma'_1, \sigma'_2, \sigma'_3, \sigma'_4\} \xrightarrow{\ell_2} \{\sigma''_1, \sigma''_2, \sigma''_3\}\}}_{=\bar{\mathsf{t}}_4}.$$

Clearly, $\{\bar{\mathsf{t}}_2\} \bar{\subseteq}_\succ \{\bar{\mathsf{t}}_4\}$ and $\{\bar{\mathsf{t}}_3\} \bar{\subseteq}_\succ \{\bar{\mathsf{t}}_4\}$. Furthermore, each $\bar{\mathsf{T}}$ such that $\{\bar{\mathsf{t}}_2\} \bar{\subseteq}_\succ \bar{\mathsf{T}}$ and $\{\bar{\mathsf{t}}_3\} \bar{\subseteq}_\succ \bar{\mathsf{T}}$ is also $\{\bar{\mathsf{t}}_4\} \bar{\subseteq}_\succ \bar{\mathsf{T}}$. Thus $\{\bar{\mathsf{t}}_4\}$ is the smallest set of collecting traces larger than $\{\bar{\mathsf{t}}_2\}$ and $\{\bar{\mathsf{t}}_3\}$. This result can be generalized, so that $\bar{\cup}_\succ$ can be shown to be the complete join on the abstract domain $\langle \bar{\mathsf{D}}_\succ, \bar{\subseteq}_\succ \rangle$:

LEMMA 4.2 *The operator* $\bar{\cup}_\succ$ *is a complete join on* $\langle \bar{\mathsf{D}}_\succ, \bar{\subseteq}_\succ \rangle$.

Proof. Let us consider two arbitrary elements of $\bar{\mathsf{D}}_\succ$, say $\bar{\mathsf{T}}_1$ and $\bar{\mathsf{T}}_2$. The first thing to show is that $\bar{\mathsf{T}} = \bar{\mathsf{T}}_1 \bar{\cup}_\succ \bar{\mathsf{T}}_2$ is an upper bound. By definition for each trace $\bar{\mathsf{t}}$ in $\bar{\mathsf{T}}_1$ or $\bar{\mathsf{T}}_2$ there exists a trace $\bar{\mathsf{t}}'$ that is the only one in $\bar{\mathsf{T}}$ with the same interaction history of $\bar{\mathsf{t}}$. Furthermore the states of $\bar{\mathsf{t}}'$ form a superset of the corresponding states in $\bar{\mathsf{t}}$. Therefore, by Definition 4.3 $\bar{\mathsf{T}}$ is larger than $\bar{\mathsf{T}}_1$ and $\bar{\mathsf{T}}_2$. Furthermore $\bar{\mathsf{T}}$ is the least upper bound as if $\bar{\mathsf{T}}_1 \bar{\subseteq}_\succ \bar{\mathsf{T}}'$ and $\bar{\mathsf{T}}_2 \bar{\subseteq}_\succ \bar{\mathsf{T}}'$ then for each $\bar{\mathsf{t}} \in \bar{\mathsf{T}}_1 \cup \bar{\mathsf{T}}_2 . \exists \bar{\mathsf{t}}' \in \bar{\mathsf{T}}' . \bar{\mathsf{t}} \bar{\subseteq}^\tau_\succ \bar{\mathsf{t}}'$. By definition of $\bar{\subseteq}^\tau_\succ$, it means that history$(\bar{\mathsf{t}})$ = history$(\bar{\mathsf{t}}')$ and for all the $i \geq 0 . \bar{\mathsf{t}}(i) = S_i$ and $\bar{\mathsf{t}}'(i) = S'_i$ then $S_i \subseteq S'_i$. By definition of $\bar{\cup}_\succ$, it exists a trace $\bar{\mathsf{t}}_\cup$ in $\bar{\mathsf{T}}$ such that history$(\bar{\mathsf{t}})$ = history$(\bar{\mathsf{t}}_\cup)$. Furthermore for all $i \geq 0 . \bar{\mathsf{t}}(i) \subseteq \bar{\mathsf{t}}_\cup(i)$. Thus $\bar{\mathsf{T}} \bar{\subseteq}_\succ \bar{\mathsf{T}}'$, so that $\bar{\cup}_\succ$ is a join morphism. The completeness follows directly from the properties of union of sets and the uniqueness of traces with the same interaction history.

q.e.d.

It is worth noting that if $\bar{\mathsf{T}}_1$ and $\bar{\mathsf{T}}_2$ are two elements of $\bar{\mathsf{D}}_\succ$ that do not contain traces with the same interaction history then $\bar{\mathsf{T}}_1 \bar{\cup}_\succ \bar{\mathsf{T}}_2$ reduces to set union. This is formalized by the following lemma:

LEMMA 4.3 *Let* $\bar{\mathsf{T}}_1$ *and* $\bar{\mathsf{T}}_2$ *be two elements of* $\bar{\mathsf{D}}_\succ$. *If*

$$\forall \bar{\mathsf{t}}_1 \in \bar{\mathsf{T}}_1 . \forall \bar{\mathsf{t}}_2 \in \bar{\mathsf{T}}_2 .\ \mathrm{history}(\bar{\mathsf{t}}_1) \neq \mathrm{history}(\bar{\mathsf{t}}_2)$$

then $\bar{\mathsf{T}}_1 \bar{\cup}_\succ \bar{\mathsf{T}}_2 = \bar{\mathsf{T}}_1 \cup \bar{\mathsf{T}}_2$.

Proof. Using the hypothesis, it is immediate to see that in Definition 4.4: $(\bar{\mathsf{T}}_1 \cup \bar{\mathsf{T}}_2)/_{\mathrm{history}} = \{\{\bar{\mathsf{t}}\} \mid \bar{\mathsf{t}} \in \bar{\mathsf{T}}_1 \vee \bar{\mathsf{t}} \in \bar{\mathsf{T}}_2\} = \{\bar{\mathsf{t}} \in \bar{\mathsf{T}}_1\} \cup \{\bar{\mathsf{t}} \in \bar{\mathsf{T}}_2\}$. Thus, $\bar{\mathsf{T}}_1 \bar{\cup}_\succ \bar{\mathsf{T}}_2 = \bar{\mathsf{T}}_1 \cup \bar{\mathsf{T}}_2$.

q.e.d.

The meet operator can be defined dually w.r.t. the join. Furthermore, following the lines of the above proof it can be shown to be a complete meet:

DEFINITION 4.5 (MEET, $\bar{\cap}_\succ$) *Let* $\bar{\mathsf{T}}_1$ *and* $\bar{\mathsf{T}}_2$ *be two elements of* $\bar{\mathsf{D}}_\succ$, *then the meet* $\bar{\cap}_\succ \in [\bar{\mathsf{D}}_\succ \times \bar{\mathsf{D}}_\succ \to \bar{\mathsf{D}}_\succ]$ *is*

$$\bar{\mathsf{T}}_1 \bar{\cap}_\succ \bar{\mathsf{T}}_2 = \{\bar{\cap}^\tau_\succ \bar{\mathsf{T}} \mid \bar{\mathsf{T}} \in (\bar{\mathsf{T}}_1 \cap \bar{\mathsf{T}}_2)_{/\mathrm{history}}\},$$

where $\bar{\cap}^\tau_\succ \in [\mathcal{T}(\mathcal{P}(\Sigma)) \times \mathcal{T}(\mathcal{P}(\Sigma)) \to \mathcal{T}(\mathcal{P}(\Sigma))]$ *is defined as follows:*

$$S_1 \bar{\cap}^\tau_\succ S_2 = S_1 \cap S_2$$
$$S_1 \xrightarrow{\ell} \tau_1 \bar{\cap}^\tau_\succ S_2 \xrightarrow{\ell} \tau_2 = S_1 \cap S_2 \xrightarrow{\ell} (\tau_1 \bar{\cap}^\tau_\succ \tau_2).$$

The definition of the abstract domain of collecting traces is a consequence of what has been said so far. In particular an easy consequence of the above lemmata is that it is a complete lattice:

THEOREM 4.1 (ABSTRACT DOMAIN OF COLLECTING TRACES) *The abstract domain* $\langle \bar{\mathsf{D}}_\succ, \bar{\subseteq}_\succ, \bar{\bot}_\succ, \bar{\top}_\succ, \bar{\cup}_\succ, \bar{\cap}_\succ \rangle$ *is a complete lattice.*

For the moment, the term *abstract* domain is not fully justified. In fact we have to show that the above domain is related to the domain of traces by a Galois connection. This is the goal of the next section.

4.2.2 Abstraction

The idea of the abstraction function is to collect together, at a given time, all the states that are consequence of the invocation of the same method or of the same context interaction. The formal definition is given below.

The abstraction of a state is the singleton that contains such a state. The abstraction of a set of traces T is a little bit more complicated. As for the direct interactions are concerned, we consider the set $\mathsf{T}_{\mathtt{m}}$ of all the traces in T whose first action is the invocation of the method $\mathtt{m}$. Differently stated, $\mathsf{T}_{\mathtt{m}}$ is the set of the *consequences* of $\mathtt{m}$. In order to obtain the set of collecting traces $\bar{\mathsf{T}}'$ we recursively apply the abstraction function to the traces that are consequence of an invocation of $\mathtt{m}$. The initial states of the traces in T and $\bar{\mathsf{T}}'$ are respectively S and S'. Roughly, S is the set of internal states *before* the invocation of the method $\mathtt{m}$ and S' is the set of internal states *after* the invocation of the method $\mathtt{m}$ with any input value. Then, we replace the initial states of the traces in $\bar{\mathsf{T}}'$ with S', obtaining the set of traces $\bar{\mathsf{T}}''$. Finally, the abstraction of the set of traces $\bar{\mathsf{T}}$ is a set of traces in the form $S \xrightarrow{\mathtt{m}} S' \to \tau_c$, where $S' \to \tau_c$ belongs to $\bar{\mathsf{T}}''$. The case of indirect interaction is similar.

Definition 4.6 (Traces Abstraction, $\alpha_\succ$) *The abstraction function* $\alpha_\succ \in [\mathcal{P}(\mathcal{T}(\Sigma)) \to \bar{\mathsf{D}}_\succ]$ *is defined as follows:*

$$
\begin{aligned}
&\alpha_\succ(S) = S \qquad\qquad \text{if } S \subseteq \Sigma\\
&\alpha_\succ(\mathsf{T}) = \{S \xrightarrow{\mathtt{m}} \tau_c \mid \ \mathtt{m} \in \mathtt{M} \cup \{\mathtt{init}\}, \mathsf{T}_{\mathtt{m}} = \{\mathsf{t} \in \mathsf{T} \mid \mathsf{t} = \sigma \xrightarrow{\langle \mathtt{m},\mathtt{v}\rangle} \tau\},\\
&\qquad\qquad \bar{\mathsf{T}}' = \alpha_\succ(\{\tau \mid \sigma \xrightarrow{\langle \mathtt{m},\mathtt{v}\rangle} \tau \in \mathsf{T}_{\mathtt{m}}\}),\\
&\qquad\qquad S = \bigcup_{\sigma \xrightarrow{\mathtt{m}} \tau \in \mathsf{T}_{\mathtt{m}}} \{\sigma\}, S' = \bigcup_{S'' \xrightarrow{\ell} \tau \in \bar{\mathsf{T}}'} S'',\\
&\qquad\qquad \tau_c \in \bar{\mathsf{T}}'' = \{S' \xrightarrow{\mathtt{m}} \tau \mid \exists \bar{\mathsf{t}} \in \bar{\mathsf{T}}'.\ \bar{\mathsf{t}} = S''' \xrightarrow{\ell} \tau\}\}\\
&\quad \cup \{S \xrightarrow{\kappa} \tau_c \mid \mathsf{T}_\kappa = \{\mathsf{t} \in \mathsf{T} \mid \mathsf{t} = \sigma \xrightarrow{\kappa} \tau\},\\
&\qquad\qquad \bar{\mathsf{T}}' = \alpha_\succ(\{\tau \mid \sigma \xrightarrow{\kappa} \tau \in \mathsf{T}_\kappa\}),\\
&\qquad\qquad S = \bigcup_{\sigma \xrightarrow{\kappa} \tau \in \mathsf{T}_\kappa} \{\sigma\}, S' = \bigcup_{S'' \xrightarrow{\ell} \tau \in \bar{\mathsf{T}}'} S'',\\
&\qquad\qquad \tau_c \in \bar{\mathsf{T}}'' = \{S' \xrightarrow{\kappa} \tau \mid \exists \bar{\mathsf{t}} \in \bar{\mathsf{T}}'.\ \bar{\mathsf{t}} = S''' \xrightarrow{\ell} \tau\}\}.
\end{aligned}
$$

Example 4.5 Let us apply the above definition to the traces of Figure 4.1.

The set of traces to abstract is

$$\mathsf{T} = \left\{ \begin{array}{l} \sigma_1 \xrightarrow{\langle \mathtt{m_i}, \mathtt{v_1} \rangle} \sigma_2 \xrightarrow{\langle \mathtt{m_j}, \mathtt{v_2} \rangle} \sigma_3 \\ \sigma_1' \xrightarrow{\langle \mathtt{m_i}, \mathtt{v_1'} \rangle} \sigma_2' \xrightarrow{\langle \mathtt{m_j}, \mathtt{v_2'} \rangle} \sigma_3' \\ \sigma_1'' \xrightarrow{\langle \mathtt{m_i}, \mathtt{v_1''} \rangle} \sigma_2'' \xrightarrow{\kappa} \sigma_3'' \end{array} \right\}.$$

The corresponding collecting traces, abstraction of T are given by:

$$\begin{aligned} \alpha_{\succ}(\mathsf{T}) = \left\{ \begin{array}{l} \{\sigma_1, \sigma_1', \sigma_1''\} \xrightarrow{\mathtt{m_i}} \{\sigma_2, \sigma_2', \sigma_2''\} \xrightarrow{\mathtt{m_j}} \{\sigma_3, \sigma_3'\} \\ \{\sigma_1, \sigma_1', \sigma_1''\} \xrightarrow{\mathtt{m_i}} \{\sigma_2, \sigma_2', \sigma_2''\} \xrightarrow{\kappa} \{\sigma_3''\} \end{array} \right| \\ \mathsf{T}_{\mathtt{m_i}} = \mathsf{T}, \bar{\mathsf{T}}' = \alpha_{\succ} \left(\left\{ \begin{array}{l} \sigma_2 \xrightarrow{\langle \mathtt{m_j}, \mathtt{v_2} \rangle} \sigma_3 \\ \sigma_2' \xrightarrow{\langle \mathtt{m_j}, \mathtt{v_2'} \rangle} \sigma_3' \\ \sigma_2'' \xrightarrow{\kappa} \sigma_3'' \end{array} \right\} \right), \\ S = \{\sigma_1, \sigma_1', \sigma_1''\}, S' = \{\sigma_2, \sigma_2', \sigma_2''\}, \\ \bar{\mathsf{T}}'' = \left\{ \left\{ \begin{array}{l} \{\sigma_2, \sigma_2', \sigma_2''\} \xrightarrow{\mathtt{m_j}} \{\sigma_3, \sigma_3'\}, \\ \{\sigma_2, \sigma_2', \sigma_2''\} \xrightarrow{\kappa} \{\sigma_3''\} \end{array} \right\} \right\}. \end{aligned}$$

In the above formula, the function $\alpha_{\succ}$ is called recursively on a set of *shorter* traces in order to obtain T'. The details of this invocation are given below:

$$\begin{aligned} \alpha_{\succ} \left(\left\{ \begin{array}{l} \sigma_2 \xrightarrow{\langle \mathtt{m_j}, \mathtt{v_2} \rangle} \sigma_3 \\ \sigma_2' \xrightarrow{\langle \mathtt{m_j}, \mathtt{v_2'} \rangle} \sigma_3' \\ \sigma_2'' \xrightarrow{\kappa} \sigma_3'' \end{array} \right\} \right) = \left\{ \{\sigma_2, \sigma_2'\} \xrightarrow{\mathtt{m_j}} \{\sigma_3, \sigma_3'\} \right| \\ \mathsf{T}_{\mathtt{m_j}} = \left\{ \begin{array}{l} \sigma_2 \xrightarrow{\langle \mathtt{m_j}, \mathtt{v_2} \rangle} \sigma_3 \\ \sigma_2' \xrightarrow{\langle \mathtt{m_j}, \mathtt{v_2'} \rangle} \sigma_3' \end{array} \right\}, \\ \mathsf{T}' = \alpha_{\succ}(\{\sigma_3, \sigma_3'\}) = \{\sigma_3, \sigma_3'\}, \\ S = \{\sigma_2, \sigma_2'\}, S' = \{\sigma_3, \sigma_3'\}, \mathsf{T}'' = \{\sigma_3, \sigma_3'\}\} \\ \cup \{\{\sigma_2''\} \xrightarrow{\kappa} \{\sigma_3''\} \mid \mathsf{T}_\kappa = \{\sigma_2'' \xrightarrow{\kappa} \sigma_3''\}, \bar{\mathsf{T}}' = \{\sigma_3''\}, \\ S = \{\sigma_2''\}, S' = \{\sigma_3''\}, \bar{\mathsf{T}}'' = \{\sigma_3''\}\}. \end{aligned}$$

The concretization function $\gamma_{\succ}$ maps a set of collecting traces to interaction traces. In particular, the concretization of a collecting trace $\bar{\mathsf{t}}$ is the set of all the traces that have the same interaction history and whose states are taken from the nodes of $\bar{\mathsf{t}}$.

$$
\begin{aligned}
\mathsf{T}' = &\left\{ \begin{array}{ccc} \sigma_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma_3 & , & \sigma_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma'_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma_3 \\ \sigma_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma'_3 & , & \sigma_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma'_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma'_3 \end{array} \right. \\
&\left. \begin{array}{c} \sigma_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma''_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma_3 \\ \sigma_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma''_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma'_3 \end{array} \right\} \\
\cup &\left\{ \begin{array}{ccc} \sigma'_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma_3 & , & \sigma'_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma'_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma_3 \\ \sigma'_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma'_3 & , & \sigma'_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma'_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma'_3 \end{array} \right. \\
&\left. \begin{array}{c} \sigma'_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma''_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma_3 \\ \sigma'_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma''_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma'_3 \end{array} \right\} \\
\cup &\left\{ \begin{array}{ccc} \sigma''_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma_3 & , & \sigma''_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma'_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma_3 \\ \sigma''_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma'_3 & , & \sigma''_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma'_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma'_3 \end{array} \right. \\
&\left. \begin{array}{c} \sigma''_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma''_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma_3 \\ \sigma''_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma''_2 \xrightarrow{\langle \mathtt{m_j}, \mathsf{v}_2 \rangle} \sigma'_3 \end{array} \right\} \\
\cup &\left\{ \begin{array}{ccc} \sigma_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma_2 \xrightarrow{\kappa} \sigma'_3 & \sigma'_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma_2 \xrightarrow{\kappa} \sigma'_3 & \sigma''_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma_2 \xrightarrow{\kappa} \sigma'_3 \\ \sigma_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma_2 \xrightarrow{\kappa} \sigma''_3 & , \ \sigma'_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma_2 \xrightarrow{\kappa} \sigma''_3 & , \ \sigma''_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma_2 \xrightarrow{\kappa} \sigma''_3 \\ \sigma_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma_2 \xrightarrow{\kappa} \sigma''_3 & \sigma'_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma_2 \xrightarrow{\kappa} \sigma''_3 & \sigma''_1 \xrightarrow{\langle \mathtt{m_i}, \mathsf{v}_1 \rangle} \sigma_2 \xrightarrow{\kappa} \sigma''_3 \end{array} \right\}.
\end{aligned}
$$

Figure 4.2: The set of traces T', concretization of $\alpha_{\succ}(\mathsf{T})$.

Definition 4.7 (Traces Concretization, $\gamma_{\succ}$) *The concretization map, $\gamma_{\succ} \in [\bar{\mathsf{D}}_{\succ} \to \mathcal{P}(\mathcal{T}(\Sigma))]$ is defined as follows:*

$$
\begin{aligned}
\gamma_{\succ}(\bar{\mathsf{T}}) &= \bigcup_{\bar{\mathsf{t}} \in \bar{\mathsf{T}}} \gamma^{\tau}_{\succ}(\bar{\mathsf{t}}) \\
\gamma^{\tau}_{\succ}(\bar{\mathsf{t}}) &= \{\mathsf{t} \in \mathcal{T}(\Sigma) \mid \mathrm{history}(\mathsf{t}) = \mathrm{history}(\bar{\mathsf{t}}), \forall i.\ \mathsf{t}(i) \in \bar{\mathsf{t}}(i)\}.
\end{aligned}
$$

Example 4.6 Let us consider the set of collecting traces $\alpha_{\succ}(\mathsf{T})$ of Example 4.5. Then their concretization $\gamma_{\succ}(\alpha_{\succ}(\mathsf{T}))$ is the set of traces in Figure 4.2, where $\mathsf{v}_1, \mathsf{v}_2 \in \mathsf{D}_{\mathsf{in}}$.

In the above example, we can see that $\mathsf{T} \subseteq \gamma_{\succ}(\alpha_{\succ}(\mathsf{T}))$. This result always holds, so that it is possible to show that $\gamma_{\succ} \circ \alpha_{\succ}$ is extensive:

Lemma 4.4 (Extensivity of $\gamma_{\succ} \circ \alpha_{\succ}$) $\forall \mathsf{T} \in \mathcal{P}(\mathcal{T}(\Sigma)).\mathsf{T} \subseteq \gamma_{\succ}(\alpha_{\succ}(\mathsf{T}))$.

Proof. Let $\mathsf{t} \in \mathsf{T}$. Then, the definition of $\alpha_{\succ}$ implies that there exists a $\bar{\mathsf{t}} \in \alpha_{\succ}(\mathsf{T})$ that has the same history as t, i.e. $\mathrm{history}(\mathsf{t}) = \mathrm{history}(\bar{\mathsf{t}})$, so that each node i is such that $\mathsf{t}(i) \in \bar{\mathsf{t}}(i)$. By definition 4.7, $\mathsf{t} \in \gamma^{\tau}_{\succ}(\bar{\mathsf{t}})$. The thesis follows from the fact that t is an arbitrary element of T.

q.e.d.

Furthermore, it is possible to show that $\alpha_\succ \circ \gamma_\succ$ is reductive (Lemma 4.5 below), so that $\langle \alpha_\succ, \gamma_\succ \rangle$ is a Galois connection between the concrete domain of traces and the abstract domain of collecting traces (Theorem 4.2):

LEMMA 4.5 (REDUCTIVITY OF $\alpha_\succ \circ \gamma_\succ$) $\forall \bar{\mathsf{T}}.\alpha_\succ(\gamma_\succ(\bar{\mathsf{T}})) \bar{\subseteq}_\succ \bar{\mathsf{T}}$.

Proof. Let $\mathsf{t} \in \gamma_\succ(\bar{\mathsf{T}})$. By definition, there exists $\bar{\mathsf{t}} \in \bar{\mathsf{T}}$ such that $\mathsf{t} \in \gamma^\tau_\succ(\bar{\mathsf{t}})$. Then, the history of t and $\bar{\mathsf{t}}$ are the same and for each node i the state $\mathsf{t}(i)$ belongs to $\bar{\mathsf{t}}(i)$. By definition of $\alpha_\succ$, it is immediate to see that $\alpha_\succ(\{\mathsf{t}\}) \bar{\subseteq}_\succ \{\bar{\mathsf{t}}\}$. The thesis follows by considering the union on all the t's.

q.e.d.

THEOREM 4.2 (COLLECTING TRACES ABSTRACT DOMAIN) *The concrete domain of set of traces is linked to the abstract domain of sets of collecting traces by a Galois connection:*

$$\langle \mathcal{P}(\mathcal{T}(\Sigma)), \subseteq, \emptyset, \mathcal{T}(\Sigma), \cup, \cap \rangle \underset{\alpha_\succ}{\overset{\gamma_\succ}{\leftrightarrows}} \langle \bar{\mathsf{D}}_\succ, \bar{\subseteq}_\succ, \bar{\bot}_\succ, \bar{\top}_\succ, \bar{\cup}_\succ, \bar{\cap}_\succ \rangle.$$

Proof. We use Theorem 2.4. Lemma 4.4 and Lemma 4.5 state extensivity and reductivity. So, we are left to show that $\alpha_\succ$ and $\gamma_\succ$ are monotonic. But such proofs are immediate by the properties of set union.

q.e.d.

4.2.3 Abstract Semantics

We can use the abstraction $\alpha_\succ$ to design the abstract semantics by calculus. We recall from (3.1) that the semantics of an object is expressed in fixpoint form as $\mathfrak{o}[\![\mathsf{o}]\!] = \mathrm{lfp}\lambda T.\mathsf{F}(T)$, for a suitable set transformer F. Thanks to the fixpoint transfer theorem (cf. Theorem 2.6), if we have an abstract functor $\bar{\mathsf{F}}$ that satisfies the commutative property $\alpha_\succ \circ \mathsf{F}(T) \bar{\subseteq}_\succ \bar{\mathsf{F}} \circ \alpha_\succ(T)$, then

$$\alpha_\succ(\mathrm{lfp}\lambda T.\mathsf{F}(T)) \bar{\subseteq}_\succ \mathrm{lfp}\lambda T.\bar{\mathsf{F}}(T).$$

Then, the soundness of the abstract semantics is assured. In the rest of this section we design $\bar{\mathsf{F}}$ by calculus. We begin with some definitions.

The semantics of a method can be lifted to set of states in a straightforward fashion:

Definition 4.8 (Collecting Method Semantics, $\mathbb{M}[\![\cdot]\!]$) *Let* $\mathtt{m}$ *be a method and* $S \in \mathcal{P}(\Sigma)$ *a set of interaction states. Then the collecting method semantics,* $\mathbb{M}[\![\mathtt{m}]\!] \in [\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)]$ *is defined as:*

$$\begin{aligned}\mathbb{M}[\![\mathtt{m}]\!](S) = \{\langle \mathsf{e}', \mathsf{s}', \mathsf{v}', \mathsf{Esc}'\rangle \mid & \langle \mathsf{e}, \mathsf{s}, \mathsf{v}, \mathsf{Esc}\rangle \in S, \mathsf{v}_{\mathsf{in}} \in \mathsf{D}_{\mathsf{in}},\\ & \mathtt{m}[\![\mathtt{m}]\!](\mathsf{v}_{\mathsf{in}}, \mathsf{e}, \mathsf{s}) \ni \langle \mathsf{v}', \mathsf{e}', \mathsf{s}'\rangle,\\ & \mathsf{Esc}' = \mathsf{Esc} \cup \mathrm{reachable}(\mathsf{v}', \mathsf{s}')\}.\end{aligned}$$

The previous definition can be used to lift the transition function next to sets of interaction states. In particular, we recall from the previous chapter that next is made up of two parts: one concerning the direct interactions and the other concerning the indirect ones:

$$\mathrm{next}(\sigma) = \mathrm{next}_{\mathrm{dir}}(\sigma) \cup \mathrm{next}_{\mathrm{ind}}(\sigma).$$

Therefore in order to lift next to set of states, we consider its two components independently. The lifted version of the direct interaction transition function, $\mathrm{Next}_{\mathrm{dir}}$, takes as input a set of interaction states. Then for each $\mathtt{m}$ of the class, it systematically applies $\mathbb{M}[\![\mathtt{m}]\!]$ to its input. Roughly, it means that it considers all the states resulting from an invocation of the method $\mathtt{m}$ for any possible input value.

Definition 4.9 ($\mathrm{Next}_{\mathrm{dir}}$) *Let* $S \in \mathcal{P}(\Sigma)$ *be a set of interaction states. Then the function* $\mathrm{Next}_{\mathrm{dir}} \in [\mathcal{P}(\Sigma) \to \mathcal{P}(\mathcal{P}(\Sigma) \times \mathtt{M})]$ *is defined as follows:*

$$\mathrm{Next}_{\mathrm{dir}}(S) = \{\langle S', \mathtt{m}\rangle \mid \mathtt{m} \in \mathtt{M}, \mathbb{M}[\![\mathtt{m}]\!](S) = S'\}.$$

The lifted version of the indirect interaction, $\mathrm{Next}_{\mathrm{ind}}$, returns a pair made up of a set of states S' and the tag κ. The states in S' are the states as after an arbitrary modification of the values *exposed* by the object:

Definition 4.10 ($\mathrm{Next}_{\mathrm{ind}}$) *Let* $S \in \mathcal{P}(\Sigma)$ *be a set of interaction states. Then the function* $\mathrm{Next}_{\mathrm{ind}} \in [\mathcal{P}(\Sigma) \to \mathcal{P}(\mathcal{P}(\Sigma) \times \{\kappa\})]$ *is defined as follows:*

$$\mathrm{Next}_{\mathrm{ind}}(S) = \{\langle S', \kappa\rangle \mid S' = \{\sigma' \mid \exists \sigma \in S.\ \mathrm{next}_{\mathrm{ind}}(\sigma) \ni \langle \sigma', \kappa\rangle\}\}.$$

Finally, the lifted version of the transition function Next is given below:

Definition 4.11 (Next) *Let* $S \in \mathcal{P}(\Sigma)$ *be a set of interaction states. Then the transition function* $\mathrm{Next} \in [\mathcal{P}(\Sigma) \to \mathcal{P}(\mathcal{P}(\Sigma) \times \mathtt{M} \cup \{\kappa\})]$ *is*

$$\mathrm{Next}(S) = \mathrm{Next}_{\mathrm{dir}}(S) \cup \mathrm{Next}_{\mathrm{ind}}(S).$$

The design of the collecting traces object fixpoint semantics is done by calculus:

$$\begin{aligned}
&\dot{\alpha}_{\succ}(\lambda \mathsf{T}.S_0\langle \mathsf{v},\mathsf{s}\rangle \cup \{\tau \xrightarrow{\ell} \sigma \xrightarrow{\ell'} \sigma' \mid \tau \xrightarrow{\ell} \sigma \in \mathsf{T}, \mathrm{next}(\sigma) \ni \langle \sigma', \ell'\rangle\}) \\
=&// \text{ Properties of Galois connections and functional lifting of abstractions} \\
&\lambda \bar{\mathsf{T}}.\alpha_{\succ}(S_0\langle \mathsf{v},\mathsf{s}\rangle)\bar{\cup}_{\succ}\alpha_{\succ}(\{\tau \xrightarrow{\ell} \sigma \xrightarrow{\ell'} \sigma' \mid \tau \xrightarrow{\ell} \sigma \in \gamma_{\succ}(\bar{\mathsf{T}}), \mathrm{next}(\sigma) \ni \langle \sigma', \ell'\rangle\})) \\
=&// \text{ Definition of } \alpha_{\succ} \text{ and of next} \\
&\lambda \bar{\mathsf{T}}.S_0\langle \mathsf{v},\mathsf{s}\rangle\bar{\cup}_{\succ}\alpha_{\succ}(\{\tau \xrightarrow{\ell} \sigma \xrightarrow{\ell'} \sigma' \mid \tau \xrightarrow{\ell} \sigma \in \gamma_{\succ}(\bar{\mathsf{T}}), \mathrm{next}_{\mathrm{dir}}(\sigma) \ni \langle \sigma', \ell'\rangle\} \\
&\qquad\qquad \cup \{\tau \xrightarrow{\ell} \sigma \xrightarrow{\kappa} \sigma' \mid \tau \xrightarrow{\ell} \sigma \in \gamma_{\succ}(\bar{\mathsf{T}}), \mathrm{next}_{\mathrm{ind}}(\sigma) \ni \langle \sigma', \kappa\rangle\}) \\
=&// \text{ Properties of Galois connections (join morphism)} \\
&\lambda \bar{\mathsf{T}}.S_0\langle \mathsf{v},\mathsf{s}\rangle\bar{\cup}_{\succ}\alpha_{\succ}(\{\tau \xrightarrow{\ell} \sigma \xrightarrow{\ell'} \sigma' \mid \tau \xrightarrow{\ell} \sigma \in \gamma_{\succ}(\bar{\mathsf{T}}), \mathrm{next}_{\mathrm{dir}}(\sigma) \ni \langle \sigma', \ell'\rangle\}) \\
&\qquad\qquad \bar{\cup}_{\succ}\alpha_{\succ}(\{\tau \xrightarrow{\ell} \sigma \xrightarrow{\kappa} \sigma' \mid \tau \xrightarrow{\ell} \sigma \in \gamma_{\succ}(\bar{\mathsf{T}}), \mathrm{next}_{\mathrm{ind}}(\sigma) \ni \langle \sigma', \kappa\rangle\}) \\
\bar{\subseteq}_{\succ}&// \text{ Definition of } \alpha_{\succ}, \mathrm{Next}_{\mathrm{dir}} \text{ and } \mathrm{Next}_{\mathrm{ind}} \\
&\lambda \bar{\mathsf{T}}.S_0\langle \mathsf{v},\mathsf{s}\rangle\bar{\cup}_{\succ}\{\bar{\mathsf{t}} \xrightarrow{\ell} S \xrightarrow{\ell'} S' \mid \bar{\mathsf{t}} \xrightarrow{\ell} S \in \bar{\mathsf{T}}, \mathrm{Next}_{\mathrm{dir}}(S) \ni \langle S', \ell'\rangle\} \\
&\qquad\qquad \bar{\cup}_{\succ}\{\bar{\mathsf{t}} \xrightarrow{\ell} S \xrightarrow{\kappa} S' \mid \bar{\mathsf{t}} \xrightarrow{\ell} S \in \bar{\mathsf{T}}, \mathrm{Next}_{\mathrm{ind}}(S) \ni \langle S', \kappa\rangle\}
\end{aligned}$$

Finally, it is worth noting that the collecting traces above are in the form $T\bar{\cup}_{\succ}T_1\bar{\cup}_{\succ}T_2$, where T, T_1 and T_2 do not have traces that share the same interaction history. Therefore, the hypotheses of Lemma 4.3 are verified, so that using it and the fixpoint transfer theorem we can conclude that:

Theorem 4.3 (Collecting Traces Object Fixpoint Semantics,$\bar{\mathbb{O}}_{\succ}[\![\mathsf{o}]\!]$) *The abstract object semantics* $\bar{\mathbb{O}}_{\succ}[\![\mathsf{o}]\!] \in [\mathsf{D}_{\mathsf{in}} \times \mathsf{Store} \to \bar{\mathsf{D}}_{\succ}]$ *defined as*

$$\bar{\mathbb{O}}_{\succ}[\![\mathsf{o}]\!](\mathsf{v},\mathsf{s}) = \mathrm{lfp}^{\bar{\subseteq}_{\succ}}_{\bar{\perp}_{\succ}} \lambda \bar{\mathsf{T}}.S_0\langle \mathsf{v},\mathsf{s}\rangle \cup \{\bar{\mathsf{t}} \xrightarrow{\ell} S \xrightarrow{\ell'} S' \mid \bar{\mathsf{t}} \xrightarrow{\ell} S \in \bar{\mathsf{T}}, \mathrm{Next}(S) \ni \langle S', \ell'\rangle\}$$

is a sound approximation of the concrete object semantics, i.e.

$$\alpha_{\succ}(\mathbb{o}[\![\mathsf{o}]\!](\mathsf{v},\mathsf{s}))\bar{\subseteq}_{\succ}\bar{\mathbb{O}}_{\succ}[\![\mathsf{o}]\!](\mathsf{v},\mathsf{s}).$$

Using the Theorem 3.2, the extension of the previous soundness result to the class semantics is straightforward:

Corollary 4.1 (Collecting Traces Class Fixpoint Semantics,$\bar{\mathbb{C}}_{\succ}[\![\mathtt{A}]\!]$) *The abstract class semantics* $\bar{\mathbb{C}}_{\succ}[\![\mathtt{A}]\!] \in \bar{\mathsf{D}}_{\succ}$ *defined as*

$$\bar{\mathbb{C}}_{\succ}[\![\mathtt{A}]\!] = \bar{\bigcup}_{\succ}\{\bar{\mathbb{O}}_{\succ}[\![\mathsf{o}]\!](\mathsf{v},\mathsf{s}) \mid \mathsf{v} \in \mathsf{D}_{\mathsf{in}},\ \mathsf{s} \in \mathsf{Store}\}$$

is a sound approximation of the concrete class semantics, i.e. $\alpha_{\succ}(\mathbb{c}[\![\mathtt{A}]\!])\bar{\subseteq}_{\succ}\bar{\mathbb{C}}_{\succ}[\![\mathtt{A}]\!]$.

$$\begin{array}{l}\{\sigma_1,\sigma_1',\sigma_1''\} \xrightarrow{\mathtt{m_i}} \{\sigma_2,\sigma_2',\sigma_2''\} \xrightarrow{\mathtt{m_j}} \{\sigma_3,\sigma_3'\} \\ \{\sigma_1,\sigma_1',\sigma_1''\} \xrightarrow{\mathtt{m_i}} \{\sigma_2,\sigma_2',\sigma_2''\} \xrightarrow{\kappa} \{\sigma_3''\}\end{array} \quad \stackrel{\alpha_\circ}{\Longrightarrow} \left\{ \begin{array}{l}\sigma_1,\sigma_1',\sigma_1'', \\ \sigma_2,\sigma_2',\sigma_2'', \\ \sigma_3,\sigma_3',\sigma_3''\end{array} \right\}$$

Figure 4.3: The second abstraction

4.3 Second Abstraction: Reachable States

We now proceed with the second abstraction that roughly *forgets* the trace histories, keeping just the states reachable in a computation. For example, let us consider the collecting traces in Figure 4.3. The collecting traces on the left are abstract by their states so that for instance the fact that σ_2 is a consequence of the execution of the method $\mathtt{m_i}$ from one of initial states σ_1, σ_1' or σ_1'' is lost. On the other hand, the states reached in the computation are safely approximated, so that the state-based properties are preserved. As a consequence the abstraction is painless as far as we are interested in approximating the values that the fields of the objects can take, and e.g. not their evolution over the time.

4.3.1 Abstract Domain

For what said above, an element of the abstract domain is a set of states that over-approximates the states reached during a computation. As a consequence the order between the elements boils down to the subset inclusion, so that the smallest element is the empty set and the largest is Σ, the set of all the states. The join and the meet in the abstract domain are the set union and the set intersection. Finally, the abstract domain of collecting states is the complete boolean lattice

$$\langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle.$$

4.3.2 Abstraction

The abstract function $\alpha_\circ$ abstracts away from the computation history, collecting all the interaction states in the traces, *forgetting* the casual relations between sets of states. Formally it can be defined as follows:

Definition 4.12 (Collecting Traces Abstraction, $\alpha_\circ$) *The abstraction* $\alpha_\circ \in [\bar{\mathsf{D}}_\succ \to \mathcal{P}(\Sigma)]$ *is defined as follows:*

$$\alpha_\circ(\bar{\mathsf{T}}) = \bigcup_{\bar{\mathsf{t}} \in \bar{\mathsf{T}}} \alpha_\circ^\tau(\bar{\mathsf{t}}).$$

where $\alpha^{\tau}_{\circ} \in [\mathcal{T}(\mathcal{P}(\Sigma)) \to \mathcal{P}(\Sigma)]$ *is the abstraction of single collecting traces defined as follows:*

$$\alpha^{\tau}_{\circ}(S) = S \qquad \text{if } S \subseteq \Sigma$$
$$\alpha^{\tau}_{\circ}(S \xrightarrow{\ell} \tau) = S \cup \alpha^{\tau}_{\circ}(\tau).$$

Example 4.7 Applying the above definitions to the collecting traces of the Example 4.5 it is immediate to see that the abstraction function collects the states of the different traces, i.e. :

$$\mathsf{S} = \alpha_{\circ}(\alpha_{\succ}(\mathsf{T})) = \left\{ \begin{array}{l} \sigma_1, \sigma'_1, \sigma''_1, \\ \sigma_2, \sigma'_2, \sigma''_2, \\ \sigma_3, \sigma'_3, \sigma''_3 \end{array} \right\}.$$

Lemma 4.6 ($\alpha_{\circ}$ is a complete Join-morphism) *The function* $\alpha_{\circ}$ *is a complete join-morphism, i.e. for any set of abstract elements* $\{\bar{\mathsf{T}}_i\}$*:*

$$\alpha_{\circ}\left(\bar{\bigcup}_{\succ}\{\bar{\mathsf{T}}_i\}\right) = \bigcup_i \{\alpha_{\circ}(\bar{\mathsf{T}}_i)\}.$$

Proof. Let $\{\bar{\mathsf{T}}_i\}$ be a set of elements of $\bar{\mathsf{D}}_{\succ}$. Then, applying the definition of $\alpha_{\circ}$ and observing that $\bar{\cup}_{\succ}$ does not *lose* states nor it does *introduce* new ones, it follows that:

$$\alpha_{\circ}\left(\bar{\bigcup}_{\succ}\{\bar{\mathsf{T}}_i\}\right) = \bigcup_{\bar{\mathsf{t}} \in \bar{\bigcup}_{\succ}\{\bar{\mathsf{T}}_i\}} \alpha^{\tau}_{\circ}(\bar{\mathsf{t}}) = \bigcup_i \bigcup_{\bar{\mathsf{t}} \in \bar{\mathsf{T}}_i} \alpha^{\tau}_{\circ}(\bar{\mathsf{t}}) = \bigcup_i \{\alpha_{\circ}(\bar{\mathsf{T}}_i)\}.$$

q.e.d.

The abstraction above is a complete join morphism (Lemma 4.6), so that by Lemma 2.3, Lemma 2.2 and Lemma 2.4, there exists a unique concretization function $\gamma_{\circ} = \lambda S.\ \bar{\cup}_{\succ}\{\bar{\mathsf{T}} \mid \alpha_{\circ}(\bar{\mathsf{T}}) \subseteq S\}$ such that the collecting traces domain and the reachable states domain are linked by a Galois connection:

$$\langle \bar{\mathsf{D}}_{\succ}, \bar{\subseteq}_{\succ}, \bar{\bot}_{\succ}, \bar{\top}_{\succ}, \bar{\cup}_{\succ}, \bar{\cap}_{\succ} \rangle \underset{\alpha_{\circ}}{\overset{\gamma_{\circ}}{\leftrightarrows}} \langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle.$$

The abstraction $\alpha_{\circ}$ abstracts away the computation history and the *causal* relations between states, by just retaining the states reached during the computation. Therefore, the concretization of a set of states S consists of all the traces with any possible interaction history such that their nodes are identically equal to S:

$$\gamma_{\circ}(\mathsf{S}) = \bigcup_{h \in \{\mathtt{M} \cup \{\kappa\}\}^*} \{\bar{\mathsf{t}} \mid \text{history}(\bar{\mathsf{t}}) = h, \forall i.\ \bar{\mathsf{t}}(i) = \mathsf{S}\}.$$

EXAMPLE 4.8 The concretization of the set of states S of Example 4.7 is

$$\gamma_\circ(\mathsf{S}) = \left\{ \begin{array}{llll} \mathsf{S} & & & \\ \mathsf{S} \xrightarrow{\mathtt{m}_1} \mathsf{S}, & \mathsf{S} \xrightarrow{\mathtt{m}_2} \mathsf{S}, & \mathsf{S} \xrightarrow{\mathtt{m}_3} \mathsf{S}, & \ldots \\ \mathsf{S} \xrightarrow{\mathtt{m}_1} \mathsf{S} \xrightarrow{\mathtt{m}_1} \mathsf{S}, & \mathsf{S} \xrightarrow{\mathtt{m}_1} \mathsf{S} \xrightarrow{\mathtt{m}_2} \mathsf{S}, & \mathsf{S} \xrightarrow{\mathtt{m}_1} \mathsf{S} \xrightarrow{\mathtt{m}_3} \mathsf{S}, & \ldots \\ \ldots & & & \end{array} \right\}.$$

It is immediate to see that $\alpha_\succ(\mathsf{T}) \bar{\subseteq}_\succ \gamma_\circ(\mathsf{S})$.

4.3.3 Abstract Semantics

The design of the reachable states semantics is carried on by calculus. In fact, we can derive the reachable object states semantics, $\mathbb{O}[\![\mathsf{o}]\!](\mathsf{v},\mathsf{s})$, as an upper approximation of $\alpha_\circ(\bar{\mathbb{o}}_\succ[\![\mathsf{o}]\!](\mathsf{v},\mathsf{s}))$. We proceed as in the previous section, i.e. we first approximate the transfer function and then we apply the fixpoint transfer theorem to conclude the approximation of the object semantics. Before proceeding to the design-by-calculation, we need to define a function that given a set of states S returns all the possible indirect interactions between the context and the part of the internal object fields exposed to the context.

DEFINITION 4.13 (Context$(\cdot)$) *Let $S \subseteq \Sigma$ be a set of interaction states. Then their indirect interaction with the context is summarized by the function* Context$(\cdot) \in [\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)]$*:*

$$\text{Context}(S) = \{\langle \mathsf{e}, \mathsf{s}', \phi, \mathsf{Esc}\rangle \mid \langle \mathsf{e}, \mathsf{s}, \mathsf{v}, \mathsf{Esc}\rangle \in S, \exists \mathsf{a} \in \mathsf{Esc}.\ \text{update}(\mathsf{a}, \mathsf{s}) \ni \mathsf{s}'\}.$$

It is worth noting that given a set of states S, if $\langle S', \kappa\rangle \in \text{Next}_{\text{ind}}(S)$ then $S' \in \text{Context}(S)$. We exploit such a property in the derivation below:

$$
\begin{aligned}
&\dot{\alpha}_\circ(\lambda\bar{\mathsf{T}}.S_0\langle \mathsf{v},\mathsf{s}\rangle \cup \{\bar{\mathsf{t}} \xrightarrow{\ell} S \xrightarrow{\ell'} S' \mid \bar{\mathsf{t}} \xrightarrow{\ell} S \in \bar{\mathsf{T}}, \mathrm{Next}(S) \ni \langle S', \ell'\rangle\}) \\
=&//\text{Definition of Next} \\
&\dot{\alpha}_\circ(\lambda\bar{\mathsf{T}}.S_0\langle \mathsf{v},\mathsf{s}\rangle \bar{\cup}_\succ \{\bar{\mathsf{t}} \xrightarrow{\ell} S \xrightarrow{\ell'} S' \mid \bar{\mathsf{t}} \xrightarrow{\ell} S \in \bar{\mathsf{T}}, \mathrm{Next}_{\mathrm{dir}}(S) \ni \langle S', \ell'\rangle\} \\
&\qquad\qquad \bar{\cup}_\succ \{\bar{\mathsf{t}} \xrightarrow{\ell} S \xrightarrow{\kappa} S' \mid \bar{\mathsf{t}} \xrightarrow{\ell} S \in \bar{\mathsf{T}}, \mathrm{Next}_{\mathrm{ind}}(S) \ni \langle S', \kappa\rangle\}) \\
=&//\text{Definition of functional lifting of abstractions and Galois connections} \\
&\lambda S.\alpha_\circ(S_0\langle \mathsf{v},\mathsf{s}\rangle) \cup \alpha_\circ(\{\bar{\mathsf{t}} \xrightarrow{\ell} S \xrightarrow{\ell'} S' \mid \bar{\mathsf{t}} \xrightarrow{\ell} S \in \gamma_\circ(S), \mathrm{Next}_{\mathrm{dir}}(S) \ni \langle S', \ell'\rangle\}) \\
&\qquad\qquad \cup \alpha_\circ(\{\bar{\mathsf{t}} \xrightarrow{\ell} S \xrightarrow{\kappa} S' \mid \bar{\mathsf{t}} \xrightarrow{\ell} S \in \gamma_\circ(S), \mathrm{Next}_{\mathrm{ind}}(S) \ni \langle S', \kappa\rangle\}) \\
=&//\text{Definition of } \alpha_\circ, \text{ Next, Next}_{\mathrm{ind}} \text{ and Context}(\cdot) \\
&\lambda S.S_0\langle \mathsf{v},\mathsf{s}\rangle \cup \alpha_\circ(\{\bar{\mathsf{t}} \xrightarrow{\ell} S \xrightarrow{\mathtt{m}} S' \mid \bar{\mathsf{t}} \xrightarrow{\ell} S \in \gamma_\circ(S), \mathtt{m} \in \mathtt{M}, \mathbb{M}[\![\mathtt{m}]\!](S) = S'\}) \\
&\qquad\qquad \cup \alpha_\circ(\{\bar{\mathsf{t}} \xrightarrow{\ell} S \xrightarrow{\kappa} S' \mid \bar{\mathsf{t}} \xrightarrow{\ell} S \in \gamma_\circ(S), \mathrm{Context}(S) = S'\}) \\
=&//\text{Definition of } \alpha_\circ \\
&\lambda S.S_0\langle \mathsf{v},\mathsf{s}\rangle \cup S \cup \bigcup_{\mathtt{m}\in\mathtt{M}} \mathbb{M}[\![\mathtt{m}]\!](S) \cup S \cup \mathrm{Context}(S) \\
=&//\text{Idempotence of } \cup \\
&\lambda S.S_0\langle \mathsf{v},\mathsf{s}\rangle \cup S \cup \bigcup_{\mathtt{m}\in\mathtt{M}} \mathbb{M}[\![\mathtt{m}]\!](S) \cup \mathrm{Context}(S)
\end{aligned}
$$

At this point we can state the following theorem, that gives a fixpoint characterization of the states reached for all the possible executions of an object. The proof is based on the following technical lemma:

LEMMA 4.7 *Let $f \in [\langle D, \sqsubseteq\rangle \to \langle D, \sqsubseteq\rangle]$ be a complete join-morphism. Then:*

$$\mathrm{lfp}^{\sqsubseteq}_{\perp}\lambda S.\ S_0 \sqcup S \sqcup f(S) = \mathrm{lfp}^{\sqsubseteq}_{\perp}\lambda S.\ S_0 \sqcup f(S). \tag{4.4}$$

Proof. By hypothesis f is a complete join-morphism, hence it is continuous. So are the two functions at the left and the right member of (4.4). Because of continuity, the fixpoint can be reached after at most ω steps. We prove the thesis by proceeding on induction on the iterations:

$n = 1$ the iterations for the left and the right members are $I^1 = S_0 \sqcup \perp \sqcup f(\perp)$ and $J^1 = S_0 \sqcup f(\perp)$. By definitions of $\sqcup$ and $\perp$ it is immediate to see that $I^1 = J^1$.

$n > 1$ For the inductive step we have that at the n-step of the iterations:

$$I^n = S_0 \sqcup \bigsqcup_{i \leq n} f^i(S_0) \sqcup \bigsqcup_{i \leq n} f^i(\bot)$$
$$J^n = S_0 \sqcup \bigsqcup_{i \leq n} f^i(S_0) \sqcup f^i(\bot)$$

and as it is immediate to see that for a monotonic function $f^n(\bot) = \bigsqcup_{i \leq n} f^i(\bot)$ then $I^n = J^n$.

q.e.d.

Theorem 4.4 (Reachable States Object Fixpoint Semantics,$\mathbb{O}[\![\mathtt{o}]\!]$) *The reachable states fixpoint semantics of an object* $\mathtt{o}$ *instance of a class* $\mathtt{A} = \langle \mathtt{init}, \mathtt{F}, \mathtt{M} \rangle$ *is a function* $\mathbb{O}[\![\mathtt{o}]\!] \in [\mathsf{D_{in}} \times \mathsf{Store} \to \mathcal{P}(\Sigma)]$ *defined as follows*

$$\mathbb{O}[\![\mathtt{o}]\!](\mathtt{v}, \mathtt{s}) = \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda S.(S_0\langle \mathtt{v}, \mathtt{s} \rangle \cup \bigcup_{\mathtt{m} \in \mathtt{M}} \mathbb{M}[\![\mathtt{m}]\!](S) \cup \mathrm{Context}(S)). \tag{4.5}$$

Furthermore, $\mathbb{O}[\![\mathtt{o}]\!]$ *is a sound approximation of the concrete semantics, i.e.*

$$\alpha_\circ \circ \alpha_\succ(\mathbb{o}[\![\mathtt{o}]\!](\mathtt{v}, \mathtt{s})) \subseteq \mathbb{O}[\![\mathtt{o}]\!](\mathtt{v}, \mathtt{s}).$$

Proof. Applying the fixpoint transfer theorem to the equation derived above we obtain that

$$\alpha_\circ(\bar{\mathbb{o}}_\succ[\![\mathtt{o}]\!](\mathtt{v}, \mathtt{s})) = \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda S.(S_0\langle \mathtt{v}, \mathtt{s} \rangle \cup S \cup \bigcup_{\mathtt{m} \in \mathtt{M}} \mathbb{M}[\![\mathtt{m}]\!](S) \cup \mathrm{Context}(S)).$$

At its turn, it is immediate to see that the hypotheses of the Lemma 4.7 are fulfilled, so that S on the right side of the equality above can be dropped, proving the first part of the thesis. Finally, the soundness w.r.t. the concrete semantics is a consequence of the compositional property of Galois connections and of Theorem 4.3.

q.e.d.

The equation (4.5) captures the intuition that the states reached during the execution of an object are given by initial states (i.e. $S_0\langle \mathtt{v}, \mathtt{s} \rangle$), by the states consequence of the execution of a method (i.e. $\mathbb{M}[\![\mathtt{m}]\!]$) and by the states that are consequence of an interaction with the context (i.e. Context).

An immediate corollary to the theorem above is the soundness of the reachable states class semantics:

Corollary 4.2 (Reachable States Class Fixpoint Semantics,$\mathbb{C}[\![\mathtt{A}]\!]$)
The reachable states semantics of a class $\mathtt{A}$*:* $\mathbb{C}[\![\mathtt{A}]\!] \in \mathcal{P}(\Sigma)$ *defined as*

$$\mathbb{C}[\![\mathtt{A}]\!] = \bigcup\{\mathbb{O}[\![\mathtt{o}]\!](\mathsf{v},\mathsf{s}) \mid \mathtt{o} \textit{ is an instance of } \mathtt{A}, \mathsf{v} \in \mathsf{D_{in}}, \mathsf{s} \in \mathsf{Store}\}$$

is a sound approximation of the class concrete semantics, i.e.

$$\alpha_{\circ} \circ \alpha_{\succ}(\mathbb{c}[\![\mathtt{A}]\!]) \subseteq \mathbb{C}[\![\mathtt{A}]\!].$$

Furthermore, it can be expressed in fixpoint form as:

$$\mathbb{C}[\![\mathtt{A}]\!] = \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda S.(\mathbb{I}[\![\mathtt{init}]\!] \cup \bigcup_{\mathtt{m} \in \mathtt{M}} \mathbb{M}[\![\mathtt{m}]\!](S) \cup \mathrm{Context}(S)), \tag{4.6}$$

where $\mathbb{I}[\![\mathtt{init}]\!] \in \mathcal{P}(\Sigma)$ *are the states reached after any invocation of the class constructor:*

$$\mathbb{I}[\![\mathtt{init}]\!] = \bigcup_{\langle \mathsf{v},\mathsf{s}\rangle \in \mathsf{D_{in}} \times \mathsf{Store}} S_0\langle \mathsf{v},\mathsf{s}\rangle.$$

Proof. The first point of the corollary is an immediate consequence of the definition of the class semantics and of the soundness of the stepwise design of the abstract semantics. The fixpoint formulation of the reachable states semantics of class $\mathtt{A}$ is a consequence of Theorem 3.2.

q.e.d.

Chapter 5

Inference of Class Invariants

> Omnium rerum principia parva sunt. [1]
>
> Marcus Tullius Cicero,
> *De Finibus* (45 BCE)

A class invariant is a property valid for all the class instances, before and after the execution of any method. The goal of this chapter is to introduce a generic framework for the automatic and modular inference of sound class invariants for object oriented languages.

Using the fixpoint formulation of the class reachable states, the most precise state-based class invariant can be expressed as a solution of a system of equations involving the class constructor and methods. Nevertheless, in general such an invariant may be not computable. As a consequence, an abstraction is required in order to derive an effective algorithm for the inference of class invariants. In particular, we show how a class invariant can be iteratively computed on the top of a static analysis of the class constructor and methods. Thus, if the provided static analysis is a sound approximation of the semantics of the methods then the consequent class invariant is a sound approximation of the class semantics. We discuss the modularity in the derivation of the class invariant as well as the complexity issues for the fixpoint computation.

This chapter is based on the published paper [77].

[1] (Latin) The beginnings of all the things are always small.

5.1 Overview of Class Invariants

A class is correct or incorrect not by itself, but with respect to a specification. For instance, a specification can be the absence of runtime errors, such as null-pointers dereference or the absence of uncaught exceptions. More generally, a specification can be expressed in a suitable formal language. The software engineering community [81] proposes to annotate the source code with class invariants, method preconditions and postconditions in order to specify the *desired* behavior of the class.

The natural question with such an approach is: *"Does the class respect its specification?"*. The traditional approach is to monitor the assertions, so that for instance the preconditions and the class invariant are checked before the execution of a method. Such an approach has many drawbacks. For example, it requires checking arbitrary complex assertions so that it may introduce a non-negligible slowdown at runtime. Moreover it is inherently not sound. In fact the code must be executed in order to test if an assertion is violated or not. However, program execution or testing can only cover finitely many test cases so that the global validity of the assertion cannot be proved. Therefore the need for formal methods arises.

5.2 Class Invariants in the Literature

Many authors pointed out the importance of class invariants for the verification of object oriented programs and hence of the *components-based* approach to the software development [85]. We can distinguish two main lines of thought in the verification of object oriented languages, that we name the *top-down* and the *bottom-up* one.

According to the top-down approach to the verification of components, e.g, [81, 67], the class source code is annotated with a *specification* that looks like a contract between the class instances and their clients. This specification consists of a "candidate" class invariant and "candidate" preconditions and postconditions. The conformity of the class to its "candidate" invariants is either checked at runtime or by means of proof assistants.

The second approach, i.e. the bottom-up [46, 44, 47], is somehow complementary to the first one. The goal of such an approach is to infer the behavior of the class from its semantics, if necessary making use of code annotations. Then the inferred information is checked against the class specification, for example to prove that runtime errors do not occur.

In the rest of this section we briefly review some existing top-down and bottom-up approaches.

5.2.1 Design by Contract

Under the *Design by Contract* (DbC) theory [81], a software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations, i.e. contracts. In such a view, a class is annotated with an invariant and each method is endowed with a precondition and a postcondition. The annotations can be used as a specification for the clients of the components and their validity can be checked at runtime.

The most relevant incarnation of the DbC philosophy is the Eiffel programming language [80]. It provides explicit language constructs for the specification of "contracts". The Eiffel compiler can be instructed to introduce into the compiled code runtime checks for the assertions. In fact, according to a compilation switch [48], one can have:

- `CHECK_NO`: disable all checkings. It corresponds to the *release* builds;
- `CHECK_REQUIRE`: check the preconditions for each method;
- `CHECK_ENSURE`: check the postconditions for each method;
- `CHECK_INVARIANT`: check the class invariant before and after each method is called;
- `CHECK_ALL`: check any assertions.

Nevertheless, the Eiffel compiler approach has some drawbacks. First, the annotations are not guaranteed to hold for all the instantiation contexts, so that they can hardly be defined *invariants*. Second, the behavior of a program depends on the compilation switch: e.g. in the release build a class may violate its annotations without any warning be raised. On the other hand, if `CHECK_INVARIANT` is enabled then such a violation causes the program to fail. Third, the annotation language is a large subset of the Eiffel expressions. As a consequence, the assertions to be checked can be arbitrarily complex and they can introduce side-effects. Fourth, the class annotations are hand-made, so that they do not necessarily reflect the implementation of the class. Fifth, even when the `CHECK_ALL` switch is specified some care must be paid for the handling of function callbacks, as a naive approach may lead to unsound conclusions [61].

5.2.2 Java Modeling Language

The Java Modeling Language (JML) [67] is a formal behavioral interface specification language for Java. It allows to specify both the syntactic interface of the Java code and its behavior. The syntactic interface consists of

names, visibility, type checking informations, etc. of a class. The behavior interface describes the semantic behavior of a class as the preconditions and the postconditions of methods.

JML adds to the Eiffel approach to Design by Contract paradigm the expressiveness of model-oriented specification languages *à la* Larch [58]. For instance, the JML assertion language endows Java expressions with quantifiers.

There exist several tools that made use of the JML. For example the JML compiler `jml` is an extension to the Java compiler that compiles JML-annotated programs including the runtime checks of the annotations in the bytecode. The LOOP tool [63] translates JML annotations into proof obligations that are feed to a proof assistant to verify the soundness of classes w.r.t. the JML specification. The problem with such an approach is that the use of a proof assistant is tedious and it requires an interaction with the user.

5.2.3 Assertions in Java and .net

Last generation mainstream object oriented languages as Java and object oriented frameworks as .net provide a minimal support for DbC through runtime assertions.

The version 1.4 of the Java platform [56] introduced the support to assertion facilities through the keyword `assert`, the `AssertionError` class and a few additional methods to `java.lang.ClassLoader`. The command `assert bExp : str` checks if the boolean expression `bExp` evaluates to `true`. If not, the program fails and prints `str` on the error stream. Such a facility can be used to implement a very simple form of DbC, in particular to check method preconditions.

The .net framework [82], through the class `Debug`, provides a set of methods and properties that helps the code debugging. In particular the method `Debug.Assert` checks the boolean condition passed as a parameter, and if it is not satisfied, then it displays a dialog box with an error message. This facility is available to all the languages that support the .net architecture as C# [84] or the compilers for such platform as Visual C++ .Net [83]. Thus, it is quite easy to write a class that on the top of `Debug` provides a form of DbC and that exploiting the inter-language features of .net is available to all the compilers for the platform.

5.2.4 Daikon

The Daikon tool [44] is an invariant-like detector for C, C++ and Java. The tool idea is to infer pseudo-invariants from a set of execution traces: first,

the analyzing program is executed on a set of sensible inputs and then the execution traces are matched against a list of *wished* properties. If some property in this list is valid for all the tested traces, then it is said to be a pseudo-invariant. The advantage of such an approach is that it is relatively easy to implement w.r.t. a complete static analyzer or theorem prover and that it can be easily parallelized, each execution test being executed on a different computer. Nevertheless it is inherently unsound, as the test can cover just a finite number of executions. Furthermore, there are some kind of properties, e.g. most of those related to parallelism, that cannot handled by such an approach. For instance, possible deadlocks or sharing violations are unlikely to be discovered with such methods, as it relies too heavily on effective executions.

5.2.5 ESC/Java and Houdini

The Extended Static Checker for Java (ESC/Java) [47] is a tool for statically and modularly finding errors in Java programs. It is based on user annotations, that supply the preconditions, the postconditions and the class invariants. Then it uses a theorem prover in order to verify that the annotations are both consistent with the program semantics and they do not raise runtime exceptions. A drawback of such an approach is that modularity is obtained through annotations, that specify the behavior of other modules, so that they can be either incomplete or inconsistent w.r.t. the program semantics.

Houdini [46] is an *annotation assistant* developed in order to overcome this problem. The idea of the tool is to generate the annotations of a class and then to use ESC/Java to see whether the generated annotations are consistent with its semantics. If they are then the generated annotations are invariants for the class. Otherwise, the set of annotations is refined and the process is iterated. Such an approach corresponds to a greatest fixpoint computation, the starting point being the initial set of annotations. This implies that the soundness and the precision of the induced invariants depend on the initial set of annotations.

5.2.6 Some Static Analyses for Object Oriented Languages

Several static analyses have been developed for object-oriented languages with different goals. For instance, the analysis of Zee and Rinard, [105], focuses on the removal of write-barriers in generational garbage collectors; the analysis of Pollet, Le Charlier and Cortesi, [92], infers the "may" and "must" relations in Java programs; the analysis of Blanchet, [8], determines

whether the lifetime of objects exceedes or not their static scope; the analysis of Spoto and Jensen, [99], proposes a hierarchy of class analyses obstained as an abstraction of the trace semantics.

Some static analyses for object-oriented languages present modular features in that they analyze a program fragment without requiring the full program to stay in memory. For instance, Chatterjee, Ryder and Landi, [18], propose a modular alias analysis for C++ programs and Probst, [93], presents a modular control-flow analysis for object-oriented languages. Nevertheless those analyses are different from our work in that both are not able to discover class invariants, essentially because they exploit modularity at method, and not class, level.

The problem of inferring *specialized* class invariants has been faced by [3, 51] and [40]. The aim of [3] and [51] is to optimize Java programs by removing array-bounds checks. The presented analyses are capable to infer class invariants in the form of

$$\texttt{a} == \texttt{null} \vee \ 0 \leq \texttt{b} \leq \texttt{a.length},$$

where `a` is an array and `b` an integer. On the other hand, Detlefs, [40], discusses the importance of the inference of class invariants for optimizing the performances of the Java's garbage collector. In particular, the author sketchs a static analysis for determining when the reference counters of an heap-allocated object is equal to zero, so that it can be removed from the memory. Our framework for the inference of class invariants is more general than those, in that it can handle more generic properties. Furthermore we prove the soundness of our approach w.r.t. a concrete semantics.

5.3 Automatic Inference of Class Invariants

We present an approach to the verification of object-oriented programs based on abstract interpretation. It consists in computing an approximation of the class semantics and checking whether it satisfies the specification. In particular, if a sound class invariant, which is inferred from the program source, matches the specification then the class itself matches the specification, because of soundness. Therefore a static analyzer capable of inferring sound class invariants can be used as an effective verification tool. Furthermore automatically inferred class invariants can be used to optimize the compiled-code, e.g. to drop superfluous exception handlers or synchronizations, and for code documentation.

5.3.1 Strongest State-based Class Invariant

The equations that characterize a class invariant can be derived directly from the fixpoint formulation of the class reachable states of Corollary 4.2. In fact, according to such a result, the reachable states of a class $\mathtt{A} = \langle \mathtt{init}, \mathtt{F}, \mathtt{M} \rangle$ are given by:

$$\mathbb{C}[\![\mathtt{A}]\!] = \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda S.(\mathbb{I}[\![\mathtt{init}]\!] \cup \bigcup_{\mathtt{m} \in \mathtt{M}} \mathbb{M}[\![\mathtt{m}]\!](S) \cup \mathrm{Context}(S)). \tag{5.1}$$

Recalling the definition of the lfp operator, (5.1) denotes the least solution w.r.t. set inclusion larger than the empty set of the following recursive equation:

$$S = \mathbb{I}[\![\mathtt{init}]\!] \cup \bigcup_{\mathtt{m} \in \mathtt{M}} \mathbb{M}[\![\mathtt{m}]\!](S) \cup \mathrm{Context}(S).$$

In its turn, such an equation can be rewritten as the following system of recursive equations, where n is the number of methods of $\mathtt{A}$:

$$\begin{aligned} S &= S_0 \cup \bigcup_{1 \leq i \leq n} S_i \cup \mathrm{Context}(S) \\ S_0 &= \mathbb{I}[\![\mathtt{init}]\!] \\ S_i &= \mathbb{M}[\![\mathtt{m_i}]\!](S) \qquad\qquad 1 \leq i \leq n. \end{aligned} \tag{5.2}$$

A solution of the above system of equations is a tuple of sets of states $\langle \mathsf{S}, \mathsf{S_0}, \mathsf{S_1}, \ldots \mathsf{S_n} \rangle$ where

- S are the states reached after the execution of a class constructor and at the entry and exit point of any method in the class;
- $\mathsf{S_0}$ are the states reached after the execution of a class constructor;
- $\mathsf{S_i}$ are the states reached after the execution of the method $\mathtt{m_i}$.

Stated in another way around, S is a class invariant, $\mathsf{S_0}$ is a constructor postcondition, and each $\mathsf{S_i}$ is a postcondition of the corresponding method $\mathtt{m_i}$. The least, w.r.t. set inclusion, solution of such a system is $\langle \mathbb{C}[\![\mathtt{A}]\!], \mathbb{I}[\![\mathtt{init}]\!], \mathbb{M}[\![\mathtt{m_i}]\!](\mathbb{C}[\![\mathtt{A}]\!]) \ldots \mathbb{M}[\![\mathtt{m_n}]\!](\mathbb{C}[\![\mathtt{A}]\!]) \rangle$ (Lemma 5.1). It represents the strongest state-based class invariant, the strongest constructor postcondition and the strongest postconditions of the methods:

Lemma 5.1 (Strongest State-based Class Property) *The tuple of sets of states*

$$\langle \mathbb{C}[\![\mathtt{A}]\!], \mathbb{I}[\![\mathtt{init}]\!], \mathbb{M}[\![\mathtt{m_i}]\!](\mathbb{C}[\![\mathtt{A}]\!]) \ldots \mathbb{M}[\![\mathtt{m_n}]\!](\mathbb{C}[\![\mathtt{A}]\!]) \rangle$$

is the least solution of (5.2) *w.r.t. pointwise set-inclusion on* $\mathcal{P}(\Sigma)^{n+2}$. *Hence it is the strongest class-based property.*

Proof. Let $\langle \mathsf{S}, \mathsf{S}_0, \mathsf{S}_1, \ldots \mathsf{S}_n \rangle \in \mathcal{P}(\Sigma)^{n+2}$ be a solution of (5.2). By (5.1), $\mathbb{C}[\![\mathtt{A}]\!] \subseteq \mathsf{S}$. Furthermore, it is trivial to check that $\mathbb{M}[\![\mathtt{m_i}]\!]$ is monotonic, so that $\mathbb{C}[\![\mathtt{A}]\!] \subseteq \mathsf{S}$ implies that $\mathbb{M}[\![\mathtt{m_i}]\!](\mathbb{C}[\![\mathtt{A}]\!]) \subseteq \mathbb{M}[\![\mathtt{m_i}]\!](\mathsf{S}) = \mathsf{S_i}$. Finally, as the tuple is a solution of the system of equations, then $\mathbb{I}[\![\mathtt{init}]\!] = \mathsf{S}_0$ necessarily.

q.e.d.

In general the strongest state-based class property is not computable so that we need to perform an abstraction in order to safely approximate it.

5.3.2 Abstraction

The framework we present is highly generic: it is language independent and more importantly any abstract domain can be plugged in. As abstract domains consider particular properties (e.g. pointer aliasing or linear relationships between variables) the choice of a particular domain influences the property reflected by the so-inferred class invariant. Hence it influences the check of the program specification. For instance, if the specification is, in some formal language, *"The class never raises the* `null`*-pointer exception"* then we are likely to instantiate the framework with a pointer analysis, in order to show that the methods in the class never cause the throwing of such an exception.

Let $\langle \bar{\mathsf{D}}, \bar{\sqsubseteq}, \bar{\bot}, \bar{\top}, \bar{\sqcup}, \bar{\sqcap} \rangle$ be an abstract domain approximating sets of interaction states, i.e. it is linked to the concrete domain by a Galois connection:

$$\langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{\mathsf{D}}, \bar{\sqsubseteq}, \bar{\bot}, \bar{\top}, \bar{\sqcup}, \bar{\sqcap} \rangle. \tag{5.3}$$

Moreover, let us consider the abstract counterpart for the constructor collecting semantics so that the initial states are approximated by a function $\bar{\mathbb{I}}[\![\mathtt{init}]\!] \in \bar{\mathsf{D}}$ such that:

$$\mathbb{I}[\![\mathtt{init}]\!] \subseteq \gamma(\bar{\mathbb{I}}[\![\mathtt{init}]\!]). \tag{5.4}$$

The collecting method semantics of a method $\mathtt{m_i}$ of $\mathtt{A}$ is approximated by an abstract semantic function $\bar{\mathbb{M}}[\![\mathtt{m_i}]\!] \in [\bar{\mathsf{D}} \to \bar{\mathsf{D}}]$. Roughly, $\bar{\mathbb{M}}[\![\mathtt{m_i}]\!]$ is a computable and sound approximation of the semantics of the method $\mathtt{m_i}$:

$$\forall \mathsf{S} \in \mathcal{P}(\Sigma).\ \mathbb{M}[\![\mathtt{m_i}]\!](\mathsf{S}) \subseteq \gamma(\bar{\mathbb{M}}[\![\mathtt{m_i}]\!](\alpha(\mathsf{S}))). \tag{5.5}$$

From previous chapters we know that the context is free to change the internal state exposed by an object and that this behavior is captured by the function

Context. The abstract counterpart of such a function is $\overline{\text{Context}} \in [\bar{\mathsf{D}} \to \bar{\mathsf{D}}]$, defined as follows:

$$\forall \mathsf{S} \in \mathcal{P}(\Sigma).\ \text{Context}(\mathsf{S}) \subseteq \gamma(\overline{\text{Context}}(\alpha(\mathsf{S}))). \tag{5.6}$$

Finally it is possible to state the following theorem, that gives a characterization of class invariants and postconditions as solutions of a system of recursive equations on $\bar{\mathsf{D}}$ that mimics (5.2):

THEOREM 5.1 (SOUNDNESS OF CLASS INVARIANT) *Let* $\mathtt{A} = \langle \mathtt{init}, \mathtt{F}, \mathtt{M} \rangle$ *be a class,* $\bar{\mathsf{D}}$ *an abstract domain that is linked to the domain of reachable states according to* (5.3)*. Moreover, let*

- $\bar{\mathbb{I}}[\![\mathtt{init}]\!] \in \bar{\mathsf{D}}$ *be a sound approximation of the initial states as in* (5.4)*;*
- $\bar{\mathbb{M}}[\![\mathtt{m_i}]\!] \in [\bar{\mathsf{D}} \to \bar{\mathsf{D}}]$ *be an abstract semantic function that satisfies the soundness condition* (5.5)*; and*
- $\overline{\text{Context}} \in [\bar{\mathsf{D}} \to \bar{\mathsf{D}}]$ *a sound approximation of the context behavior* (5.6)*.*

Then a tuple $\langle \bar{\mathsf{I}}, \bar{\mathsf{I}}_0, \bar{\mathsf{I}}_1 \ldots \bar{\mathsf{I}}_n \rangle \in \bar{\mathsf{D}}^{n+2}$*, solution of the following recursive equations system:*

$$\begin{aligned} X &= X_0 \bar{\sqcup} \bar{\bigsqcup_{1 \le i \le n}} X_i \bar{\sqcup} \overline{\text{Context}}(X) \\ X_0 &= \bar{\mathbb{I}}[\![\mathtt{init}]\!] \\ X_i &= \bar{\mathbb{M}}[\![\mathtt{m_i}]\!](X) \qquad\qquad 1 \le i \le n. \end{aligned} \tag{5.7}$$

is such that $\mathbb{C}[\![\mathtt{A}]\!] \subseteq \gamma(\bar{\mathsf{I}})$, $\mathbb{I}[\![\mathtt{init}]\!] \subseteq \gamma(\bar{\mathsf{I}}_0)$ *and for all the methods* $\mathtt{m_i}$ *of* A $\mathbb{M}[\![\mathtt{m_i}]\!](\mathbb{C}[\![\mathtt{A}]\!]) \subseteq \gamma(\bar{\mathsf{I}}_i)$.

Proof. The system of equations above is in the form $\vec{X} = \bar{\mathsf{F}}(\vec{X})$, where $\bar{\mathsf{F}}$ is a monotonic operator on $[\bar{\mathsf{D}}^{n+2} \to \bar{\mathsf{D}}^{n+2}]$. On the other hand, the system of concrete equations is in the form $\vec{X} = \mathsf{F}(\vec{X})$, where F is a monotonic operator on $[\mathcal{P}(\Sigma)^{n+2} \to \mathcal{P}(\Sigma)^{n+2}]$. By theorem hypotheses, $\bar{\mathsf{F}}$ is a sound approximation of F. By the fixpoint transfer theorem [29], the least fixpoint of $\bar{\mathsf{F}}$ is a sound approximation of the concrete properties. If $\bar{\mathsf{I}}$ is a solution of (5.7), then it is larger than the least fixpoint of $\bar{\mathsf{F}}$. Then the thesis follows from the monotonicity of γ.

q.e.d.

A class invariant obtained through (5.7) is a sound approximation of the concrete class semantics:

COROLLARY 5.1 *Under the hypotheses of Theorem 5.1, if* $\bar{\mathsf{I}}$ *is a class invariant that satisfies* (5.7) *then*

$$\mathbb{c}[\![\mathtt{A}]\!] \subseteq \gamma_{\succ} \circ \gamma_{\circ} \circ \gamma(\bar{\mathsf{I}}).$$

Proof. According to Corollary 4.2 and definition of Galois connections we have that $\mathbb{c}[\![\mathtt{A}]\!] \subseteq \gamma_{\succ} \circ \gamma_{\circ}(\mathbb{C}[\![\mathtt{A}]\!])$. By Theorem 5.1 above $\mathbb{C}[\![\mathtt{A}]\!] \subseteq \gamma(\bar{\mathsf{I}})$. The thesis follows from the fact that in Galois connections the concretization function is monotonic.

q.e.d.

In Theorem 5.1 we assumed that all the methods are analyzed on the same abstract domain. Nevertheless, the result can be easily generalized to the case of different methods analyzed using different abstract domains. For example, one can think to analyze methods with few variables using a very precise, but expensive, abstract domain and the others with a less precise/cheaper abstract domain.

5.4 A Bank Account Example

As a first example let us consider a rather classical one: a Java class which implements a bank account. The definition of the class `Account`, taken from [71, §4], is in Figure 5.1. It has three fields, `acctNumber`, `balance` and `name` which store respectively the account number, the current balance and the name of the person holding such an account. According to the Design by Contract approach, the code must be annotated with an expression specifying that the value of `balance` is always non-negative. This is a *top-down* approach. On the other hand, we follow a *bottom-up* approach, in that we *infer* from the class definition that the value of balance is always non-negative.

5.4.1 Abstract Domain

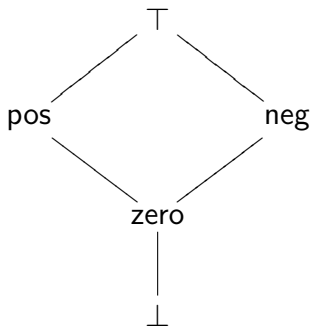
According to the abstract interpretation methodology, the first step is the design of the abstract domain for the inference of class invariants. We choose to abstract sets of interaction states with a pair $\langle \mathsf{sign}, \mathsf{Esc} \rangle$ whom first element denotes the sign of the value of `balance` and the second captures the fields that may escape the object scope. As a consequence, the abstract domain is

$$\bar{\mathsf{D}} = \mathsf{Sign} \times \mathcal{P}(\{\mathtt{acctNumber}, \mathtt{balance}, \mathtt{name}\}),$$

```
public class Account {    // Implements a Bank Account
   private long acctNumber;
   private double balance;
   private String name;
   private final double RATE = 0.045; // interest rate 4.5%

   public Account(String owner, long account, double initial) {
      name = owner;
      acctNumber = account;
      if ( initial >= 0.0 )
         balance = initial;
      else
         balance = 0.0;
   }

   public double getBalance() {
      return balance;
   }

   public double deposit(double amount) {
      if (amount < 0)
         throw new ExceptionAccount();
      else
         balance += amount;
      return balance;
   }

   public double withdraw (double amount, double fee) {
      amount += fee;
      if ((amount < 0) || (amount > balance))
         throw new ExceptionAccount();
      else
         balance -= amount;
      return balance;
   }

   public double withdrawAll(){
      double tmp = balance;
      balance = 0;
      return tmp;
   }

   public double addInterest() {
      balance += (balance * RATE);
      return balance;
   }
}
```

Figure 5.1: A Basic Bank Account Example

where Sign is the abstract domain of signs depicted below:



The order, the least and the largest element, the join and the meet operators of the abstract domain $\bar{\mathsf{D}}$ are defined in the obvious, component-wise way. The abstraction function for values, $\alpha_{\mathsf{s}} \in [\mathcal{P}(\mathsf{Val}) \to \mathsf{Sign}]$ and the corresponding concretization $\gamma_{\mathsf{s}} \in [\mathsf{Sign} \to \mathcal{P}(\mathsf{Val})]$ are given below:

$$\alpha_{\mathsf{s}}(\mathsf{V}) = \begin{cases} \bot & \text{if } \mathsf{V} = \emptyset \\ \mathsf{zero} & \text{if } \mathsf{V} = \{0\} \\ \mathsf{pos} & \text{else if } \forall \mathsf{v} \in \mathsf{V}.\ \mathsf{v} \geq 0 \\ \mathsf{neg} & \text{else if } \forall \mathsf{v} \in \mathsf{V}.\ \mathsf{v} \leq 0 \\ \top & \text{otherwise.} \end{cases} \qquad \gamma_{\mathsf{s}}(\mathsf{d}) = \begin{cases} \emptyset & \text{if } \mathsf{d} = \bot \\ \{0\} & \text{if } \mathsf{d} = \mathsf{zero} \\ \mathbb{Z}^+ \cup \{0\} & \text{if } \mathsf{d} = \mathsf{pos} \\ \mathbb{Z}^- \cup \{0\} & \text{if } \mathsf{d} = \mathsf{neg} \\ \mathsf{Val} & \text{if } \mathsf{d} = \top \end{cases}$$

The abstraction of sets of states, $\alpha \in [\mathcal{P}(\Sigma) \to \bar{\mathsf{D}}]$ is built on the top of α_{s}. Roughly, it abstracts away the field `accntNumber`, the field `name` and the value returned by a method. On the other hand, it keeps the sign of the values taken by the field `balance` and it keeps the fields stored in an address that is exposed to the context:

$$\alpha(\mathsf{S}) = \langle \alpha_{\mathsf{s}}(\{\mathsf{s}(\mathsf{e}(\texttt{balance})) \mid \langle \mathsf{e}, \mathsf{s}, \mathsf{v}, \mathsf{Esc} \rangle \in \mathsf{S}\}),\ \{\mathtt{f} \mid \exists \langle \mathsf{e}, \mathsf{s}, \mathsf{v}, \mathsf{Esc} \rangle \in \mathsf{S}.\ \mathsf{e}(\mathtt{f}) \in \mathsf{Esc}\} \rangle.$$

It is worth noting that we must keep the information on the *escaping* fields in order to infer a non-trivial invariant. If not, by the soundness requirement all the fields are assumed to escape and hence their value is undetermined. Thus the consequent class invariant is trivially the largest element of the abstract domain, i.e. "I do not know". We will return on this issue in the next section.

Once that the abstract domain is set up, the definition of the abstract counterparts for the constructor and methods semantics is straightforward. In particular as for the escaping information is concerned, we can consider a very rough escape analysis that considers the type of value returned by a

method. If the returned value belongs to a non-primitive type, then all the fields of that type may escape. If not then it is a primitive type and the Java semantics [56] assures that the field cannot escape.

5.4.2 Fixpoint Computation

A class invariant for `Account` can be obtained by instantiating the equation system (5.7) and solving it iteratively.

A first observation is that as the height of the abstract domain is finite the iterations will converge in a finite number of steps. Furthermore we ignore the getter methods, `getBalance` and `getAccountNumber`, as they do not modify the object state. According to the recursion schema (2.1), the iterations begin with the bottom element of $\bar{\mathsf{D}}^6$, that is:

$$I^0 = \langle\langle\bot,\emptyset\rangle,\langle\bot,\emptyset\rangle,\langle\bot,\emptyset\rangle,\langle\bot,\emptyset\rangle,\langle\bot,\emptyset\rangle,\langle\bot,\emptyset\rangle\rangle.$$

The first component in the vector represents the class invariant, the second one the constructor postcondition and the others the postconditions of the methods. The first iteration roughly corresponds to the abstract execution of the class constructor. In fact it is immediate to see that

$$\bar{\mathbb{I}}[\![\texttt{Account.Account}(\dots)]\!] = I^1_{\texttt{Account}} = \langle \mathsf{pos},\emptyset\rangle$$

whereas for all the methods `m` of `Account`:

$$\bar{\mathbb{M}}[\![\texttt{m}]\!](\langle\bot,\emptyset\rangle) = \langle\bot,\emptyset\rangle.$$

As a consequence, the first approximation for the class invariant is $\langle\bot,\emptyset\rangle\bar{\sqcup}\langle \mathsf{pos},\emptyset\rangle = \langle \mathsf{pos},\emptyset\rangle$, so that

$$I^1 = \langle\langle \mathsf{pos},\emptyset\rangle,\langle \mathsf{pos},\emptyset\rangle,\langle\bot,\emptyset\rangle,\langle\bot,\emptyset\rangle,\langle\bot,\emptyset\rangle,\langle\bot,\emptyset\rangle\rangle.$$

Roughly, after the creation of the object the context can decide to invoke any of the methods of the class. In this case, the methods are invoked in a state where the object fields are approximated by $\langle \mathsf{pos},\emptyset\rangle$, i.e. the approximation of the fields after the creation of the object and the execution of the class constructor. Therefore the postconditions for the methods `deposit`, `withdraw` and `addInterest` are

$$\bar{\mathbb{M}}[\![\texttt{deposit}]\!](\langle \mathsf{pos},\emptyset\rangle) = \bar{\mathbb{M}}[\![\texttt{withdraw}]\!](\langle \mathsf{pos},\emptyset\rangle) = \bar{\mathbb{M}}[\![\texttt{addInterest}]\!](\langle \mathsf{pos},\emptyset\rangle) = \langle \mathsf{pos},\emptyset\rangle.$$

On the other hand, the method `withdrawAll` resets the field `balance` so its postcondition is

$$\bar{\mathbb{M}}[\![\texttt{withdrawAll}]\!](\langle \mathsf{pos},\emptyset\rangle) = \langle \mathsf{zero},\emptyset\rangle.$$

Afterward we have to consider the possible interaction of the context on the fields that escapes the object state. Nevertheless, as `balance` does not escape from its scope there is no indirect interaction, so that $\overline{\text{Context}}(\langle \mathsf{pos}, \emptyset \rangle) = \langle \mathsf{pos}, \emptyset \rangle$. The resulting vector of invariants for the second iteration is

$$I^2 = \langle \langle \mathsf{pos}, \emptyset \rangle, \langle \mathsf{pos}, \emptyset \rangle, \langle \mathsf{pos}, \emptyset \rangle, \langle \mathsf{pos}, \emptyset \rangle, \langle \mathsf{zero}, \emptyset \rangle, \langle \mathsf{pos}, \emptyset \rangle \rangle.$$

It is immediate to see that the third iteration does not modify the vector, so that the fixpoint is reached and I^2 is a sound approximation of the class invariant and the methods postconditions.

Summing up, we showed that in any instance of `Account`, the value of `balance` is always positive or equal to zero and it does not expose its internal state to the context. Such invariants can be used for verification, as the fact that `balance` is always positive or equal to zero assures that the balance of an account can never be *red*. In particular, it assures that an invocation of the method `withdrawAll` never cause *miraculous* vanishing of debts as a balance can be only positive or equal to zero. Furthermore, they can be used for documentation: they can be shipped with the class source and they describe the behavior of the class.

5.5 Escaping Scope

The example above emphasizes that a consequence of the soundness requirement is that the abstract domain and the abstract operations must be expressive enough to capture escape information [8]. For example, if a method `m` returns a reference to an object field `f`, then the context is free to arbitrary modify the value of `f`. Because of soundness, a static analysis of `m`, $\bar{\mathbb{M}}[\![\texttt{m}]\!]$, must determine that `f` escapes its scope. Furthermore, because of the soundness of $\overline{\text{Context}}$, the corresponding class invariant $\bar{\mathsf{I}}$ will be such that $\bar{\mathsf{I}}(\texttt{f}) = \bar{\top}$, i.e. `f` may assume any value. This is the only sound assumption if we want to infer a class invariant valid for all the possible contexts.

As an example, we can consider the C++ class `Walk` below, which implements a random walk. The movement is ruled by the boolean field `goRight`: if it is true then it moves on the right, otherwise it moves on the left. Nevertheless, the class exposes `goRight` to the context through the method `getDirPointer` which returns a pointer to the field.

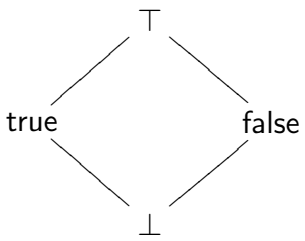
```
class Walk { // Implements a random walk

private:
  int pos;
  bool goRight;
```

```
public:
  Walk() {
    pos = 0;
    goRight = true;
  };
  void doStep() {
    if(goRight) pos += 4;
    else        pos -= 6;
  };
  bool* getDirPointer() {
    return &goRight;
  };
};
```

For the analysis of the class `Walk` we proceed as in the `Account` example. As abstract domain we chose

$$\bar{\mathsf{D}} = \mathsf{Con} \times \mathsf{Bool} \times \mathcal{P}(\{\mathtt{pos}, \mathtt{goRight}\}),$$

where Con is the abstract domain of linear congruences [57] and Bool is the simple abstract domain for booleans depicted below:



Intuitively, an element of the abstract domain is a triple $\langle \mathsf{c}, \mathsf{b}, \mathsf{E} \rangle$ where c is a linear congruence that upper-approximates the values taken by `pos`, b abstracts the value of the field `goRight` and E denotes the fields that may escape. The order on the abstract domain is a point-wise extension of the order on the underlying abstract domains. The least and the largest element as well as the join and the meet can be also defined point-wise. Once that the abstract domain is set up, the abstract semantics of the constructor and of the methods can be easily obtained.

In order to obtain an invariant for `Walk` we use the iteration schema (2.1). It is worth noting that $\bar{\mathsf{D}}$ satisfies the ACC condition, so that the iterations are assured to converge. The iterations begin with the bottom element of $\bar{\mathsf{D}}^4$:

$$I^0 = \langle\langle \bot, \bot, \emptyset\rangle, \langle \bot, \bot, \emptyset\rangle, \langle \bot, \bot, \emptyset\rangle, \langle \bot, \bot, \emptyset\rangle\rangle.$$

The first iteration roughly corresponds to the abstract execution of the class constructor, so that $\bar{\mathbb{I}}[\![\texttt{Walk} :: \texttt{Walk}()]\!] = \langle 0, \mathsf{true}, \emptyset\rangle$ implies that:

$$I^1 = \langle\langle \texttt{pos} = 0, \mathsf{true}, \emptyset\rangle, \langle \texttt{pos} = 0, \mathsf{true}, \emptyset\rangle, \langle \bot, \bot, \emptyset\rangle, \langle \bot, \bot, \emptyset\rangle\rangle.$$

Next iteration involves the method semantics. Roughly, at this point the object has been created and the context can choose between invoking the method `doStep` or `getDirPointer`. Thus, `doStep` and `getDirPointer` are invoked with the object fields approximated as in the previous iteration:

$$\begin{aligned}\bar{\mathbb{M}}[\![\texttt{doStep}]\!](\langle \texttt{pos} = 0, \mathsf{true}, \emptyset\rangle) &= \langle \texttt{pos} = 4, \mathsf{true}, \emptyset\rangle\\ \bar{\mathbb{M}}[\![\texttt{getDirPointer}]\!](\langle \texttt{pos} = 0, \mathsf{true}, \emptyset\rangle) &= \langle \texttt{pos} = 0, \mathsf{true}, \{\texttt{goRight}\}\rangle.\end{aligned}$$

It is worth noting, that the constructor does not return any value and in particular it does not expose the object internal state, so that $\overline{\text{Context}}(\langle \texttt{pos} = 0, \mathsf{true}, \emptyset\rangle)$ is trivially the identity. The approximation for the class invariant is obtained by merging the invariant at the previous step and the abstract states at the exit-point of the two methods. Therefore, the vector for the second iteration is

$$\begin{aligned}I^2 = \langle\langle \texttt{pos} \equiv 0(4), \mathsf{true}, \{\texttt{goRight}\}\rangle, \langle \texttt{pos} = 0, \mathsf{true}, \emptyset\rangle,\\ \langle \texttt{pos} = 4, \mathsf{true}, \emptyset\rangle, \langle \texttt{pos} = 0, \mathsf{true}, \{\texttt{goRight}\}\rangle\rangle.\end{aligned}$$

At the third step, we have a situation where the context can interact with the object either directly, i.e. invoking a method, or indirectly, i.e. changing the value of `goRight`:

$$\begin{aligned}\bar{\mathbb{M}}[\![\texttt{doStep}]\!](\langle \texttt{pos} \equiv 0(4), \mathsf{true}, \{\texttt{goRight}\}\rangle) &= \langle \texttt{pos} \equiv 0(4), \mathsf{true}, \{\texttt{goRight}\}\rangle\\ \bar{\mathbb{M}}[\![\texttt{getDirPointer}]\!](\langle \texttt{pos} \equiv 0(4), \mathsf{true}, \{\texttt{goRight}\}\rangle) &= \langle \texttt{pos} \equiv 0(4), \mathsf{true}, \{\texttt{goRight}\}\rangle\\ \overline{\text{Context}}(\langle \texttt{pos} \equiv 0(4), \mathsf{true}, \{\texttt{goRight}\}\rangle) &= \langle \texttt{pos} \equiv 0(4), \top, \{\texttt{goRight}\}\rangle.\end{aligned}$$

It is worth noting that after an indirect interaction, the value of the boolean field is undetermined. The consequent vector of invariants is:

$$\begin{aligned}I^3 = \langle\langle \texttt{pos} \equiv 0(4), \top, \{\texttt{goRight}\}\rangle, \langle \texttt{pos} = 0, \mathsf{true}, \emptyset\rangle,\\ \langle \texttt{pos} \equiv 0(4), \mathsf{true}, \{\texttt{goRight}\}\rangle, \langle \texttt{pos} \equiv 0(4), \mathsf{true}, \{\texttt{goRight}\}\rangle\rangle\end{aligned}$$

The iterations are still not stable, so that they continue:

$$\begin{aligned}\bar{\mathbb{M}}[\![\texttt{doStep}]\!](\langle \texttt{pos} \equiv 0(4), \top, \{\texttt{goRight}\}\rangle) &= \langle \texttt{pos} \equiv 0(2), \top, \{\texttt{goRight}\}\rangle\\ \bar{\mathbb{M}}[\![\texttt{getDirPointer}]\!](\langle \texttt{pos} \equiv 0(4), \top, \{\texttt{goRight}\}\rangle) &= \langle \texttt{pos} \equiv 0(4), \top, \{\texttt{goRight}\}\rangle\\ \overline{\text{Context}}(\langle \texttt{pos} \equiv 0(4), \top, \{\texttt{goRight}\}\rangle) &= \langle \texttt{pos} \equiv 0(4), \top, \{\texttt{goRight}\}\rangle.\end{aligned}$$

and the resulting vector is

$$I^4 = \langle\langle \texttt{pos} \equiv 0(2), \top, \{\texttt{goRight}\}\rangle, \langle \texttt{pos} = 0, \mathsf{true}, \emptyset\rangle,$$
$$\langle \texttt{pos} \equiv 0(2), \top, \{\texttt{goRight}\}\rangle, \langle \texttt{pos} \equiv 0(4), \top, \{\texttt{goRight}\}\rangle\rangle.$$

At next step the only component of the vector that changes is the one corresponding to `getDirPointer`, as

$$\bar{\mathbb{M}}[\![\texttt{getDirPointer}]\!](\langle \texttt{pos} \equiv 0(2), \top, \{\texttt{goRight}\}\rangle) = \langle \texttt{pos} \equiv 0(2), \top, \{\texttt{goRight}\}\rangle,$$

so that the vector corresponding to the fifth step is

$$I^5 = \langle\langle \texttt{pos} \equiv 0(2), \top, \{\texttt{goRight}\}\rangle, \langle \texttt{pos} = 0, \mathsf{true}, \emptyset\rangle,$$
$$\langle \texttt{pos} \equiv 0(2), \top, \{\texttt{goRight}\}\rangle, \langle \texttt{pos} \equiv 0(2), \top, \{\texttt{goRight}\}\rangle\rangle.$$

It is immediate to see that a further iteration does not change the vector, i.e. $I^5 = I^6$, so that the least fixpoint is reached.

To sum up, we proved that the value assumed by the field `pos` is, for all the possible instances and contexts of `Walk`, an even number. This result is valid even if the object expose to the context the field `goRight`, which controls the *direction* and the *length* of the step during the walk. Furthermore the invariant on `pos` that we inferred, i.e. its value is always even, is not explicitly present in the source code and it cannot be derived by a simple scan of the `Walk` source code. As a consequence, our *bottom-up* approach for the inference of class invariants is more powerful than those based on source-code and class documentation syntactic scanning as e.g. [6].

5.6 Fixpoint Computation and Complexity

In the examples above we have seen that the least solution of (5.7) can be computed using the standard fixpoint iterations. However, in general the height of the domain $\bar{\mathsf{D}}$ is infinite. In such a case, the convergence of the iterations must be forced through the use of a widening operator $\bar{\nabla} \in [\bar{\mathsf{D}} \times \bar{\mathsf{D}} \to \bar{\mathsf{D}}]$. An immediate iteration schema, whose soundness is justified by the chaotic iterations theorem (cf. Theorem 2.8), consists in mimicking the interactions between an object and a context. The first step is the creation of the object and the initialization of the fields to the invocations of the constructor. The second step consists in the non-deterministic invocation of one of the methods of the class without considering indirect interactions. In fact, as the class constructor does not return any value, in particular it cannot expose the object state so that no indirect-interaction may occur.

The further steps are the non-deterministic invocation of one of the methods of the class joined with the indirect interactions:

$$
\begin{aligned}
I^0 &= \bar{\mathbb{I}}[\![\texttt{init}]\!] \\
I^1 &= I^0 \bar{\sqcup} \bar{\bigsqcup_{1\le i\le n}} \bar{\mathbb{M}}[\![\texttt{m}_\texttt{i}]\!](I^0) \\
I^{k+1} &= I^k \bar{\sqcup} \bar{\bigsqcup_{1\le i\le n}} \bar{\mathbb{M}}[\![\texttt{m}_\texttt{i}]\!](I^k) \bar{\sqcup} \overline{\mathrm{Context}}(I^k) \qquad \text{if } 1 \le k < w \qquad (5.8) \\
I^{k+1} &= I^k \bar{\nabla} \bar{\bigsqcup_{1\le i\le n}} \bar{\mathbb{M}}[\![\texttt{m}_\texttt{i}]\!](I^k) \bar{\nabla} \overline{\mathrm{Context}}(I^k) \qquad \text{if } k \ge w
\end{aligned}
$$

The iteration schema above performs w exact iterations and then it applies the widening operator. By definition of the widening, after further $w_{\bar{\nabla}}$ steps the iterations converge to a post-fixpoint.

The immediate implementation of the iteration strategy of (5.8) is to consider at step $k+1$:

$$
\begin{aligned}
I_1^{k+1} &= \bar{\mathbb{M}}[\![\texttt{m}_1]\!](I^k) \\
&\vdots \\
I_n^{k+1} &= \bar{\mathbb{M}}[\![\texttt{m}_\texttt{n}]\!](I^k)
\end{aligned}
$$

and then to take the least upper bound, i.e. $\bar{\bigsqcup} I_i^{k+1}$. If the analysis is performed on a single sequential machine then the cost, in time or memory, is the sum of the cost of the analysis of each method: $\kappa_{\text{seq}} = \kappa_1 + \kappa_2 + \ldots \kappa_n$, where κ_i is the cost of computing $\bar{\mathbb{M}}[\![\texttt{m}_\texttt{i}]\!](I^k)$. Stated differently, κ_i is the cost of analyzing $\texttt{m}_\texttt{i}$ when the initial states are specified by I^k.

Therefore the cost of the whole analysis is proportional to the cost of analyzing the methods times the total number of iterations: $\mathcal{O}(\kappa_{\text{seq}} \cdot (w + w_{\bar{\nabla}}))$.

If we have n processing units then the computation can be performed in parallel by allocating the analysis of each $\texttt{m}_\texttt{i}$ on a different unit. In fact, the body of a method $\texttt{m}_\texttt{i}$ may invoke a method $\texttt{m}_\texttt{j}$ that belongs to the same class. However, as such a call to $\texttt{m}_\texttt{j}$ is somehow private to the class, i.e. it is not an explicit invocation of the class user, then at this point the class invariant may not hold [81]. So, the access the $\texttt{m}_\texttt{j}$ code is enough to parallelize the fixpoint computation. The soundness of the result of the computation is assured by Theorem 2.9. If enough computation resources are available, then the cost of one iteration step is the maximum of the cost of the analyses of each method: $\kappa_{\text{par}} = \max\{\kappa_1, \ldots \kappa_n\}$ so that of the whole analysis is $\mathcal{O}(\kappa_{\text{par}} \cdot (w + w_{\bar{\nabla}}))$.

From the previous considerations we can conclude that the cost of the class invariant inference is a function of the underlying static analyses of the

methods' bodies. Therefore in a practical implementation it is important to find a good trade-off between the analysis cost and the desired precision. This may depend on different factors, such as the goal of the class invariant: if it is used for verification purposes, then it is reasonable to use a high-precision/expensive static analyses. On the other hand, if the goal is the discovery of runtime errors in early stages of code development, then a not-so-precise/fast static analysis can be useful.

5.7 Modularity and Program Analysis

Our approach is modular as it satisfies the three requirements of composability, decomposability and understandability (cf. Section 1.1.3). Classes are analyzed without any hypothesis on the calling context, so that decomposability and understandability are fulfilled.

If a class `A` to be analyzed has a variable (field, method formal parameter, etc.) of type `B` then `B` must be analyzed before `A`. However the analysis of `A` does not strictly require the code of `B` and it can use the class invariant and the postconditions of the methods: any reference in `A` to an object of type `B` can be conservatively substituted by the `B` class invariant, and any invocation of a `B` method can be replaced by its postcondition. Therefore the composability requirement is satisfied.

The advantage of the approach is that if two or more distinct classes use the same class `B` then it can be analyzed once and its result can be used many times, speeding up the whole analysis. Eventually, if two or more classes depend in a cyclic way, then they must either be analyzed together or the circularity must be broken with the technique of separation of the analyses of Chapter 10.

In general an object-oriented program consists of a set of classes $\{\mathtt{A_i}\}$ and a main expression. For the moment we do not consider inheritance as it will be the subject of the Chapter 8. From the set of classes we can statically derive the (inverse) graph of the *has-A* relation [90]. This graph is such that the nodes are the classes of the program and there is an edge from `B` to `A` if and only if the class `A` has fields or local variables of type `B`. The program analysis begins with the initial nodes, i.e. the nodes that have no predecessor. Once these nodes are analyzed (if possible in parallel on distinct computation units) the successors can be considered and so on. If a cycle is found, then the considerations of the last paragraph apply. Most of the time, the analysis of a full program does not require the analysis of all the classes it uses. In fact, in general a program will use some library classes. These can be analyzed once and the result (re-) used by all the programs that use of them.

5.8 Discussion

We present a modular and generic framework for the inference of class invariants. The invariants are obtained from the program source, without any human interaction or annotation. We illustrated the approach with two examples and we discussed the complexity of the analysis. Nevertheless, such a direct approach has some drawbacks:

- the cost of the fixpoint computation. The inference of a class invariant consists of two nested fixpoint computations: one for the analysis methods and the other for the class invariants;
- the analysis of inheritance. A naive analysis of subclasses using the expansion of inheritance [90] may cause a quadratic slowdown. Furthermore, it does not take into account the sharing of computations enabled by inheritance;
- the handling of mutually recursive classes. Classes that are mutually dependent must be analyzed together, so that the encapsulation features of object-oriented languages are not fully exploited.

We face the problem of the cost in Chapters 6–7, of the analysis of inheritance in 8–9 and of mutually recursive classes in 10.

Chapter 6

Symbolic Relations for the Approximation of Set of Traces

> Everything should be made as simple as possible, but not simpler.
>
> Albert Einstein (1950)

In this chapter we introduce a generic constraint domain for relational symbolic modular analysis. The idea is that the semantics of a module can be approximated by a set of relations symbolically linking the input, output and local variables. We show how this result is correct w.r.t. a trace semantics, and how it can be used to perform an (incremental) modular analysis. Such a generic constraint domain captures the structure of a given class of modular static analyses, namely the symbolic relational ones [35], so that in a certain sense the results of this chapter are orthogonal to those of the rest of the thesis.

On the other hand, it turns out that symbolic relational analyses are worthwhile for the modular static analysis of object-oriented languages. For instance, in the next chapter we will show how they can be used to obtain a precise yet not-so-expensive class invariant and in Chapter 8 we will show how they can be used for a precise handling of down-calls and up-calls.

This chapter is based on the published paper [76].

6.1 Relational Symbolic Abstract Domains

The main idea of symbolic relational static analyses [76, 35] is that the values that input, output and local variables can simultaneously take are restrained by some constraints. In this chapter we present a generic abstract domain ($\mathcal{A}$-domain) which axiomatizes this kind of analyses. The abstract domain is made up by a set of relations, their interpretation and a constraint *simplification* operator. The soundness of the approach w.r.t. a compositional and modular trace semantics is shown.

This construction is effective in that several relation-based modular analysis as e.g. [32, 18, 91] can be directly formulated as instances of our framework. So, in a certain sense we can say that the $\mathcal{A}$-domain factors out the common structure of these analyses, and it simplifies their correctness proofs, since it is sufficient to show that they fulfill the hypotheses of being an $\mathcal{A}$-domain. Eventually, we give a hint of how to perform incremental modular analyses by refining the result of the analysis using an approximation of the values of local variables.

6.2 Module Abstraction by Relations

A natural way to abstract the module semantics is by keeping relations between the input and the output values. For example [32] has shown how linear inequalities can be used to approximate the input/output behavior of a function. We formalize and extend this idea, by giving a generic abstract domain whose intuition is to approximate a set of traces by symbolically linking the input, the output and the local environments. The so obtained result can then be used in two ways: either by dropping the local values and keeping just the relations between the input and the output, so that by instantiating the input values the output is automatically determined, or by running an imprecise, worst-case analysis on the module in order to approximate the values of local variables and then to use these values and the actual input to have a better approximation of the output.

We begin by giving the elements, the order and the operations of the abstract domain and the corresponding soundness results.

6.2.1 Constraints

The elements of our abstract domain are sets of constraints. A constraint is a relation between variables at (possibly) different program points. The idea is that variables range over a given domain and a constraint restricts the values they can simultaneously take. To exploit modular analysis, we want

to compute the interdependencies between the module input and the output.

We use the module's variables indexed by the program point they refer to. So for example if $\mathtt{x}$ is a variable at module's entry point it will be denoted as $\mathtt{x}_{(\mathtt{in})}$. Formally if V and $\mathtt{PP}$ are the sets of respectively module variables and program points then a *labeled* variable belongs to the set

$$\text{Vars} = \{\mathtt{x}_{(\mathtt{pp})} \mid \mathtt{x} \in V, \mathtt{pp} \in \mathtt{PP}\}.$$

The formal meaning of a label variable is given by a set of states, that is by a function $\Gamma \in [\text{Vars} \to \mathcal{P}(\Sigma)]$ such that

$$\Gamma(\mathtt{x}_{(\mathtt{i})}) = \{\sigma \in \Sigma \mid \sigma(\mathtt{x}) \in \text{dom}(\mathtt{x}),\ \sigma(\mathtt{pp}) = \mathtt{i}\},$$

where $\text{dom}(\mathtt{x})$ is the range domain for $\mathtt{x}$, e.g. if it is a pointer then $\text{dom}(\mathtt{x}) = \mathsf{Addr}$ and $\mathtt{pp}$ denotes the program point.

A constraint describes the interdependencies between variables, so it can be defined as a term built on Vars and a set of Terms. For example, $\mathsf{PointsTo}(\mathtt{y}_{(3)}, \mathtt{null})$ is built from the variable $\mathtt{y}_{(3)}$, the atom $\mathtt{null}$ and the constructor $\mathsf{PointsTo}$. In general in order to describe the module behavior we need finitely many constraints. Formally:

Definition 6.1 (Constraints) *Let* Vars *be a set of labeled variables,* Term *a set of terms and* Rel *the set of relations built on them, i.e.*

$$\text{Rel} = \{\rho[\mathtt{t}_1 \dots \mathtt{t}_m](\mathtt{x}_1 \dots \mathtt{x}_n) \mid \rho[\cdot] \textit{ is a relation constructor},\ \mathtt{t}_i \in \text{Terms},\ \mathtt{x}_j \in \text{Vars}\}.$$

Then the set of constraints is defined as $\mathsf{C} = \mathcal{P}_{\text{fin}}(\text{Rel})$*, and its elements are denoted by* c*.*

Moreover, we need a function that given a set of constraints returns the set of restrained variables, e.g. $\text{tiedVars}(\{\mathsf{PointsTo}(\mathtt{y}_{(3)}, \mathtt{null})\}) = \{\mathtt{y}_{(3)}\}$. So it is natural to define the function $\text{tiedVars} \in [\mathsf{C} \to \mathcal{P}(\text{Vars})]$ as

$$\text{tiedVars}(\mathsf{c}) = \bigcup_{\rho[\mathtt{t}_1 \dots \mathtt{t}_m](\mathtt{x}_1 \dots \mathtt{x}_n) \in \mathsf{c}} \{\mathtt{x}_1, \dots \mathtt{x}_n\}.$$

6.2.2 Concretization of Constraints

How is the set of constraints related to the semantics? At first we assume to have an interpretation for a set of constraints, i.e. a boolean function $\mathbb{R}[\![\cdot]\!]$ that given a n-tuple of values says if they satisfy the relation or not. Then the concretization of the constraint is the set of all the traces verifying it,

i.e. all the traces whose states are compatible with the interpretation $\mathbb{R}[\![\cdot]\!]$. Eventually, the concretization of a set of constraints is simply the intersection of concretizations. Therefore we can state the:

Definition 6.2 (Meaning Function) *Let* $\Gamma \in [\text{Vars} \to \mathcal{P}(\Sigma)]$ *be the interpretation for symbolic values,* $\mathcal{I}$ *an interpretation for terms,* D_i *the range domain of the i-th variable and* $\mathbb{R}[\![\cdot]\!] \in [\text{Rel} \to \mathcal{I} \times \mathsf{D}_1 \times \ldots \mathsf{D}_{|\text{Vars}|} \to \mathbb{B}]$ *the interpretation for constraints. Then the concretization of a single constraint* $\gamma_\rho \in [\mathsf{C} \to \mathcal{P}(\mathcal{T}(\Sigma))]$ *is defined as :*

$$
\begin{aligned}
&\gamma_\rho(\rho[\mathbf{t}](\mathtt{x}_1, \ldots \mathtt{x}_n)) = \\
&\quad \{\tau \in \mathcal{T}(\Sigma) \mid \forall \sigma_1 \ldots \sigma_n \in \alpha_\Sigma(\{\tau\}). \\
&\qquad \mathtt{x}_1 = \mathtt{y}^1_{(\mathtt{pp}_1)}, \sigma_1 \in \Gamma(\mathtt{y}^1_{(\mathtt{pp}_1)}),\ \sigma_1(\mathtt{y}^1) = v_{\mathtt{x}_1}, \sigma_1(\mathtt{pp}) = \mathtt{pp}_1, \\
&\qquad\quad \ldots \\
&\qquad \mathtt{x}_n = \mathtt{y}^n_{(\mathtt{pp}_n)}, \sigma_n \in \Gamma(\mathtt{y}^n_{(\mathtt{pp}_n)}),\ \sigma_n(\mathtt{y}^n) = v_{\mathtt{x}_n}, \sigma_n(\mathtt{pp}) = \mathtt{pp}_n \\
&\qquad \Rightarrow \mathbb{R}[\![\rho[\mathbf{t}](\mathtt{x}_1, \ldots \mathtt{x}_n)]\!](\mathcal{I}(\mathbf{t}), v_{\mathtt{x}_1}, \ldots v_{\mathtt{x}_n})\}.
\end{aligned}
$$

The concretization of a set of constraints $\gamma_\mathsf{C} \in [\mathsf{C} \to \mathcal{P}(\mathcal{T}(\Sigma))]$ *is*

$$\gamma_\mathsf{C}(\mathsf{c}) = \bigcap_{\rho[\mathbf{t}](\mathbf{x}) \in \mathsf{c}} \gamma_\rho(\rho[\mathbf{t}](\mathbf{x})).$$

With a slight abuse of terminology, in analogy with vector spaces theory, sometimes we refer to $\gamma_\mathsf{C}(\mathsf{c})$ as the space of c and to $|\text{tiedVars}(\mathsf{c})|$ as the dimension of this space.

It is worth noting that the meaning of an element in C is parameterized by the domains of the interpretation of variables and terms, i.e. the D_is and $\mathcal{I}$. Actually it turns out to be an instance of the reduced cardinal power [31, 53] construction on semantics domains. Nevertheless, the proof is out of the scopes of this work, so we skip it.

At this point we can endow C with an order. The order construction relies on the γ_C function just defined and the following technical lemma. It states that the more the constraints the smaller the space determined by them.

Lemma 6.1 (Space Reduction) *Let* $\mathsf{c}_1, \mathsf{c}_2 \in \mathsf{C}$ *such that* $\mathsf{c}_1 \subseteq \mathsf{c}_2$. *Then* $\gamma_\mathsf{C}(\mathsf{c}_1) \supseteq \gamma_\mathsf{C}(\mathsf{c}_2)$.

Proof. The hypothesis $\mathsf{c}_1 \subseteq \mathsf{c}_2$ implies that it exists a c' such that $\mathsf{c}_2 = \mathsf{c}_1 \cup \mathsf{c}'$. Then $\gamma_\mathsf{C}(\mathsf{c}_2) = \gamma_\mathsf{C}(\mathsf{c}_1 \cup \mathsf{c}') = \gamma_\mathsf{C}(\mathsf{c}_1) \cap \gamma_\mathsf{C}(\mathsf{c}') \subseteq \gamma_\mathsf{C}(\mathsf{c}_1)$. The last step is by monotonicity of intersect by enlarging gamma(c') to the maximal set (which is a unit for intersect).

q.e.d.

The $\preceq$-order arises in a natural way: it states that the more the constraints and the more the information, the less the traces:

LEMMA 6.2 (ORDER ON C) *The relation* $\preceq$ *defined as* $\forall \mathsf{c}_1, \mathsf{c}_2 \in \mathsf{C}.\ \mathsf{c}_1 \preceq \mathsf{c}_2 \Leftrightarrow \gamma_{\mathsf{C}}(\mathsf{c}_1) \subseteq \gamma_{\mathsf{C}}(\mathsf{c}_2)$ *is a preorder. Moreover, the largest element* $\curlyvee$ *is* $\emptyset$.

Proof. The proof follows immediately by the definition of the subset inclusion and the previous lemma. In particular, $\emptyset$ imposes no constraint, so that its image through γ_{C} are all the possible traces. Hence it is the largest element w.r.t. the order $\preceq$.

q.e.d.

6.2.3 Variables Dropping

We have seen that constraints relate together different variables. But what about the inverse operation, i.e. the variable elimination? In this section we introduce an axiomatic characterization of an operator δ whose goal is to eliminate a given variable from a set of constraints. Roughly speaking, we need it for two purposes: to reduce the size of constraints in order to obtain a more efficient analysis and to hide the module internal variables in the result of a module analysis.

Intuitively a drop operator δ is required to preserve the logical implication *(monotonicity)* and to lose all the information about a variable at once *(variable elimination)*. Moreover, dropping a variable causes a lose of information *(extensivity)*, the application order does not matter *(commutativity)* and dropping a variable that is not restrained has no effect *(unit)*. This is formalized by the next definition:

DEFINITION 6.3 (DROPPING OPERATOR AXIOMS) *A dropping operator on* C *is a function* $\delta \in [\mathrm{Vars} \to \langle \mathsf{C}, \preceq \rangle \to \langle \mathsf{C}, \preceq \rangle]$ *such that* $\forall \mathtt{x}, \mathtt{y} \in \mathrm{Vars}$ *and* $\forall \mathsf{c}_1, \mathsf{c}_2 \in \mathsf{C}$*:*

- $\mathsf{c}_1 \preceq \mathsf{c}_2 \Rightarrow \delta_{\mathtt{x}}(\mathsf{c}_1) \preceq \delta_{\mathtt{x}}(\mathsf{c}_2)$ *(monotonicity)*
- $\mathtt{x} \notin \mathrm{tiedVars}(\delta_{\mathtt{x}}(\mathsf{c}_1))$ *(variable elimination)*
- $\mathsf{c}_1 \preceq \delta_{\mathtt{x}}(\mathsf{c}_1)$ *(extensivity)*
- $\delta_{\mathtt{x}} \circ \delta_{\mathtt{y}}(\mathsf{c}_1) = \delta_{\mathtt{y}} \circ \delta_{\mathtt{x}}(\mathsf{c}_1)$ *(commutativity)*
- $\delta_{\mathtt{x}}(\mathsf{c}_1) = \mathsf{c}_1$, *if* $\mathtt{x} \notin \mathrm{tiedVars}(\mathsf{c}_1)$ *(unit).*

Because of commutativity property of δ we write $\delta_{\mathtt{x}_1,\mathtt{x}_2,\ldots}$ for $\delta_{\mathtt{x}_1} \circ \delta_{\mathtt{x}_2} \circ \ldots$.

A dropping operator is also idempotent (Lemma 6.3 below) so it can be seen as an abstraction in a lower-dimension domain. In fact idempotence

together with monotonicity and extensivity are the three hypotheses required for an upper closure operator [31]. Our definition is stronger in that we ask the abstraction being strict.

Some properties of δ are given by the next lemma. Briefly it states that δ is idempotent; that when dropping all the variables remains no information at all so that if the constraints are not contradictory then it corresponds to the whole set of possible traces; and that once a variable is dropped then in the concrete it assumes all the possible values of its definition domain:

LEMMA 6.3 (δ PROPERTIES) *Let δ being as in the above definition. Then* $\forall \mathtt{x} \in \text{Vars}.\ \forall \mathtt{c} \in \mathsf{C}$:

1. $\delta_{\mathtt{x}}(\mathtt{c}) = \delta_{\mathtt{x}} \circ \delta_{\mathtt{x}}(\mathtt{c})$
2. $\delta_{\mathtt{x}} \circ \delta_{\text{Vars}}(\mathtt{c}) = \delta_{\text{Vars}}(\mathtt{c})$
3. $\gamma_{\mathsf{C}}(\delta_{\text{Vars}}(\mathtt{c})) = \mathcal{T}(\Sigma)$
4. $\forall v \in \text{dom}(\mathtt{x}).\ \exists \sigma \in \alpha_{\Sigma} \circ \gamma_{\mathsf{C}} \circ \delta_{\mathtt{x}}(\mathtt{c}).\ \sigma(\mathtt{x}) = v.$

Proof. By *(variable elimination)* $\mathtt{x} \notin \text{tiedVars}(\delta_{\mathtt{x}} \circ \delta_{\mathtt{x}}(\mathtt{c}))$ so that the condition of *(unit)* is verified which implies that $\delta_{\mathtt{x}}(\mathtt{c}) = \delta_{\mathtt{x}} \circ \delta_{\mathtt{x}}(\mathtt{c})$. Idempotency implies *2.*, because by definition $\mathtt{x} \in \text{Vars}$.

As for *3.* is concerned, *(variable elimination)* implies that $\forall \mathtt{x} \in \text{Vars}.\mathtt{x} \notin \text{tiedVars}(\delta_{\text{Vars}}(\mathtt{c}))$. As a consequence, $\text{tiedVars}(\delta_{\text{Vars}}(\mathtt{c}))$ is either empty, or it contains tautologies (e.g. $1 = 1$), or it contains contradictions (e.g. $1 = 2$). Applying the definition of γ_{C}, it is immediate to verify that in all of the three cases $\gamma_{\mathsf{C}}(\emptyset) = \mathcal{T}(\Sigma)$.

Finally, if $\text{dom}(\mathtt{x})$ is empty the thesis is straightforwardly true. If not, let $v \in \text{dom}(\mathtt{x})$. By definition of δ, $\mathtt{x} \notin \text{tiedVars}(\delta_{\mathtt{x}}(\mathtt{c}))$ so that by definition of γ_{C} $\exists \tau \in \gamma_{\mathsf{C}}(\delta_{\mathtt{x}}(\mathtt{c})).\ \exists i.(\tau(i))(\mathtt{x}) = v$. The state $\tau(i)$ is the one we are looking for, and it is captured by the abstraction α_{Σ} (cfr. Example 2.1).

q.e.d.

The dual of δ, i.e. the function that drops all the variables but the specified ones will be denoted as $\pi_{\mathtt{x}_1,\mathtt{x}_2,\ldots}(\mathtt{c}) = \delta_{\text{Vars}-\{\mathtt{x}_1,\mathtt{x}_2,\ldots\}}(\mathtt{c})$.

6.2.4 Abstract Domain Operations

At this point just few operations are missing to complete the definition of our (parametric) abstract domain. Namely we need the join, the meet and a widening operator to ensure the analysis' termination.

Intuitively we need an operator $\curlyvee$ to gather the information described by its two operands so that it returns a space greater than the spaces of the two operands. Therefore it is natural to require that

$$\forall \mathsf{c}_1, \mathsf{c}_2 \in \mathsf{C}.\ \mathsf{c}_1 \preceq \mathsf{c}_1 \curlyvee \mathsf{c}_2 \text{ and } \mathsf{c}_2 \preceq \mathsf{c}_1 \curlyvee \mathsf{c}_2.$$

Nevertheless, in general we do not require $\curlyvee$ to be the *least* upper bound. An example of $\curlyvee$ is the convex-hull operation on linear inequalities.

Analogously a smaller space is obtained by adding more constraints, i.e. by putting together two sets of relations by means of the operation $\curlywedge$. So, $\forall \mathsf{c}_1,\ \mathsf{c}_2 \in \mathsf{C}$.

$$\mathsf{c}_1 \curlywedge \mathsf{c}_2 = \mathsf{c}_1 \cup \mathsf{c}_2 = \{\rho \mid \rho \in \mathsf{c}_1 \vee \rho \in \mathsf{c}_2\}.$$

Lemma 6.4 (γ_{C} is a Meet-morphism) $\forall \mathsf{c}_1, \mathsf{c}_2 \in \mathsf{C}.\ \gamma_{\mathsf{C}}(\mathsf{c}_1 \curlywedge \mathsf{c}_2) = \gamma_{\mathsf{C}}(\mathsf{c}_1) \cap \gamma_{\mathsf{C}}(\mathsf{c}_2)$.

Proof. The lemma's thesis is obtained by applying in sequence the definition of $\curlywedge$, γ_{C}, set intersection properties and the definition of γ_{C}:

$$\begin{aligned}\gamma_{\mathsf{C}}(\mathsf{c}_1 \curlywedge \mathsf{c}_2) = \gamma_{\mathsf{C}}(\mathsf{c}_1 \cup \mathsf{c}_2) &= \bigcap_{\rho \in \mathsf{c}_1 \cup \mathsf{c}_2} \gamma_\rho(\rho) \\ &= \bigcap_{\rho \in \mathsf{c}_1} \gamma_\rho(\rho) \cap \bigcap_{\rho \in \mathsf{c}_2} \gamma_\rho(\rho) = \gamma_{\mathsf{C}}(\mathsf{c}_1) \cup \gamma_{\mathsf{C}}(\mathsf{c}_2)\end{aligned}$$

q.e.d.

It is worth noting that whilst γ_{C} is a meet-morphism, in general it is not a *complete* meet-morphism. In fact, if we consider linear inequalities, then it is immediate to construct an infinite sequence $\{\mathsf{c}_i \mid i \geq 0\}$ of polyhedra which contains the unitary ball $B = \{\langle x, y\rangle \mid x^2 + y^2 \leq 1\}$, such that $\bigcap_{i\geq 0} \gamma_{\mathsf{C}}(\mathsf{c}_i) = B$ and $\gamma_{\mathsf{C}}(\bigwedge_{i\geq 0} \mathsf{c}_i)$ does not exist.

The widening takes account of the last two iterations during fixpoint computation keeping the constraints that remain stable in between. Essentially this is the idea shared by well-known abstract domains as intervals [29], polyhedra [37], octagons [86] and octahedra [20]. Formally it can be shown that the operator $\curlyvee$ is a widening in the sense of [29]:

Lemma 6.5 (Widening,$\curlyvee$) *Let* $\curlyvee \in [\mathsf{C} \times \mathsf{C} \to \mathsf{C}]$ *be defined as*

$$\curlyvee = \lambda(\mathsf{c}_1,\ \mathsf{c}_2).\ \{\rho \in \mathsf{c}_1 \mid \exists \mathsf{c}_2' \subseteq \mathsf{c}_2.\ \gamma_{\mathsf{C}}(\{\rho\}) \supseteq \gamma_{\mathsf{C}}(\mathsf{c}_2')\}.$$

Then $\curlyvee$ *is a widening.*

Proof. Let c_1 and c_2 be in C. By definition $\mathsf{c}_1 \curlyvee \mathsf{c}_2$ contains fewer constraints than both c_1 and c_2. Then, by Lemma 6.1 it is an upper bound of c_1 and c_2. Furthermore, the number of constraints can only decrease and as elements of C are finite sets of constraints then the convergence in a finite number of steps is assured.

q.e.d.

All that have been said in this section is summarized by the next definition:

DEFINITION 6.4 ($\mathcal{A}$-DOMAIN) *The abstract domain $\mathcal{A}$ is the tuple*

$$\langle \mathsf{C}_{/\equiv}, \curlyvee, \preceq, \curlyvee, \curlywedge, \barwedge, \delta \rangle$$

where the equivalence relation is defined as $\mathsf{c}_1 \equiv \mathsf{c}_2 \Leftrightarrow \gamma_{\mathsf{C}}(\mathsf{c}_1) = \gamma_{\mathsf{C}}(\mathsf{c}_2)$.

The elements of the $\mathcal{A}$-domain are related to set of traces by means of the function γ_{C}. As in general γ_{C} is not a complete meet-morphism the abstraction function may not exists. Therefore the relation between the concrete domain and the $\mathcal{A}$-domain is weaker than the traditional one based on Galois connections. As a matter of fact, we are in a more general abstract interpretation setting [34]:

LEMMA 6.6 (SOUNDNESS OF $\mathcal{A}$-DOMAIN)

$$\langle \mathcal{P}(\mathcal{T}(\Sigma)), \subseteq, \emptyset, \mathcal{T}(\Sigma), \cup, \cap \rangle \xleftarrow{\gamma_{\mathsf{C}}} \mathcal{A}\text{-}domain.$$

6.3 Analysis and Soundness

Once C has been given a domain structure by means of the results of Section 6.2 it is rather simple to analyze a module. In fact we can use the operations of an $\mathcal{A}$-domain plus some approximations of the basic constructs of the language which preserve the relations with the input.

For example, in an imperative language we need an approximation of the assignment := that do keeps the relations with the input values. Nevertheless, the assignment := is a forgetful command whose execution *deletes* some part of the knowledge we had before it. Equivalently, we can say that the effect of an assignment is twofold: to add some relations between the values before its execution and the assigned variable and to delete the information about some other variables. So the constraints of an assignment

$$(\mathtt{pp_i})\mathtt{x}{:=}\mathtt{E}(\mathtt{pp_{i+1}})$$

are given by $\rho(\mathtt{x}{:=}\mathtt{E})$, where tied variables refer to program points $\mathtt{pp_i}$ and $\mathtt{pp_{i+1}}$. Formally, if $\mathtt{PP_i}$ is the set of variables labeled with program point $\mathtt{i}$, then

$$\mathrm{tiedVars}(\rho(\mathtt{x}{:=}\mathtt{E})) \subseteq \mathtt{PP_i} \cup \mathtt{PP_{i+1}}.$$

These constraints are added to that at program point $\mathtt{pp_i}$ ($\mathsf{c}_{\mathtt{pp_i}}$) to obtain an approximation of the state after the assignment execution. Eventually just a subset V of variables is kept. So, using the operations of $\mathcal{A}$ we can state the:

LEMMA 6.7 (ASSIGNMENT SEMANTICS) *Let* $\mathbb{e}[\![\mathtt{E}]\!] \in [\Sigma \to \mathcal{P}(\Sigma)]$ *be the semantics of expression, let* $\mathsf{c}_{\mathtt{pp_i}} \in \mathsf{C}$ *and let*

$$\gamma_{\mathsf{C}}(\rho(\mathtt{x}{:=}\mathtt{E})) \supseteq \{\tau \in \mathcal{T}(\Sigma) \mid \sigma \in \alpha_{\Sigma}(\gamma_{\mathsf{C}}(\mathsf{c}_{\mathtt{pp_i}})), \sigma(\mathtt{pp}) = \mathtt{pp_i}, \\ \tau = \ldots\sigma \to \sigma' \ldots, \sigma'(\mathtt{x}) \in \mathbb{e}[\![\mathtt{E}]\!](\sigma)\}.$$

Then it is sound to define

$$\mathsf{c}_{\mathtt{pp_{i+1}}} = \rho((\mathtt{pp_i})\mathtt{x}{:=}\mathtt{E}(\mathtt{pp_{i+1}})) = \pi_{\mathrm{V}}(\rho(\mathtt{x}{:=}\mathtt{E}) \curlywedge \mathsf{c}_{\mathtt{pp_i}}).$$

Proof. (Sketch) By hypothesis $\rho(\mathtt{x}{:=}\mathtt{E})$ is a sound approximation of the assignment semantics. Therefore, $\rho(\mathtt{x}{:=}\mathtt{E}) \curlywedge \mathsf{c}_{\mathtt{pp_i}}$ approximates the traces that satisfy both constraints (before and after the execution of the assignment) and as δ is monotonic, soundness is preserved.

q.e.d.

It is worth noting that, in order to exploit modular analysis, assignment abstract semantics must (at least) keep relations with the module input values (i.e. in $\mathtt{PP_{in}}$). Then we are likely to have $\mathrm{V} \cap \mathtt{PP_{in}} \neq \emptyset$.

The previous lemma can be easily extended to cope with other basic language constructors, as sequence, conditional, etc. In particular in [76] we defined a trace semantics $\mathbb{t}[\![\mathtt{M}]\!]$ parameterized by the semantics of a set of basic operators $\{\rhd_i\}$. Then, we defined the abstract counterparts of the basic operators $\{\bar{\rhd}_i\}$ and a function $\bar{\mathbb{t}}[\![\mathtt{M}]\!]$ which mimics the trace semantics. Here we just recall the main result of [76]:

THEOREM 6.1 ($\mathcal{A}$-SOUNDNESS) *Let* M *be a module, let* $\mathbb{t}[\![\mathtt{M}]\!]$ *be a trace semantics built using a set of basic operators* $\{\rhd_i \in [\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)]\}$. *Let* $\{\bar{\rhd}_i \in [\mathsf{C} \to \mathsf{C}]\}$ *be sound counterparts of the basic operators , i.e.* $\forall i.\ \rhd_i \dot{\subseteq} \gamma_{\mathsf{C}}(\bar{\rhd}_i)$. *Then the abstract trace semantics* $\bar{\mathbb{t}}[\![\mathtt{M}]\!]$ *obtained by replacing the concrete operators with their abstract counterparts is such that*

$$\mathbb{t}[\![\mathtt{M}]\!] \dot{\subseteq} \gamma_{\mathsf{C}} \circ \bar{\mathbb{t}}[\![\mathtt{M}]\!].$$

If $\mathtt{M}$ is a program module, then $\bar{\mathfrak{t}}[\![\mathtt{M}]\!]$, can be used for a whole-program analysis. In general $\bar{\mathfrak{t}}[\![\mathtt{M}]\!]$ is in the form of $\mathsf{c}[\mathtt{x}_{\mathtt{in}}, \mathtt{x}_{\mathtt{out}}, \mathtt{x}_{\mathtt{loc}}]$ i.e. the result of the analysis keeps relations between the input, output and local variables of $\mathtt{M}$. When analyzing a program, such a relation can be used in two different ways. First, by dropping the local info, and keeping just the input/output relation:

$$\mathsf{c}'[\mathtt{x}_{\mathtt{in}}, \mathtt{x}_{\mathtt{out}}] = \delta_{\mathtt{x}_{\mathtt{loc}}}(\mathsf{c}[\mathtt{x}_{\mathtt{in}}, \mathtt{x}_{\mathtt{out}}, \mathtt{x}_{\mathtt{loc}}]).$$

It is sound because of property *(extensivity)* of droppings. Second, by computing an approximation for $\mathtt{x}_{\mathtt{loc}}$ so that in the abstract call to the module both $\mathtt{x}_{\mathtt{in}}$ and $\mathtt{x}_{\mathtt{loc}}$ can be instantiated in order to obtain a tighter result. This can be done for example by performing an imprecise and fast analysis of the module. Formally, it is possible to use a different abstract domain $\bar{\mathsf{D}}$, not necessarily relational, linked to C by a function $\mathrm{flow}^{\bar{\mathsf{D}}\to\mathsf{C}}$ to obtain $\mathsf{d}[\mathtt{x}_{\mathtt{in}}, \mathtt{x}_{\mathtt{out}}, \mathtt{x}_{\mathtt{loc}}]$. Then the final result is

$$\delta_{\mathtt{x}_{\mathtt{loc}}}(\mathsf{c}[\mathtt{x}_{\mathtt{in}}, \mathtt{x}_{\mathtt{out}}, \mathtt{x}_{\mathtt{loc}}] \curlywedge \mathrm{flow}^{\bar{\mathsf{D}}\to\mathsf{C}}(\mathsf{d}[\mathtt{x}_{\mathtt{in}}, \mathtt{x}_{\mathtt{out}}, \mathtt{x}_{\mathtt{loc}}])).$$

In Section 6.4.3 we show how a polyhedra analysis can be refined using the result of a very imprecise parity analysis.

6.4 Instantiations of the $\mathcal{A}$-domain

We prove that some well known modular analysis are instances of our $\mathcal{A}$-domain. We begin by showing how polymorphic types can be rewritten in our framework, and we give a hint of how they can be use to perform modular monotype inference:

6.4.1 Types

Intuitively the type of a function is a relation between the input and the output. This information can be expressed in our framework provided the correct instantiations. According to Definition 6.1 the terms are the types. Types are built on atoms

$$\mathrm{Types} = \mathrm{Atoms} \cup \{t_1 \to t_2,\ t_1\ \mathtt{list} \mid t_1,\ t_2 \in \mathrm{Types}\}$$

and $\mathrm{Atoms} = \{\mathtt{int},\ \mathtt{bool}\} \cup \{\alpha,\ \beta, \dots\}$. Relations are equalities in the form variable is equal to either a type or another variable, i.e.

$$\mathrm{Rel} = \{\mathtt{x} = t \mid \mathtt{x} \in \mathrm{Vars},\ t \in \mathrm{Types} \cup \mathrm{Vars}\}.$$

The interpretation function $\mathfrak{I}$ gives out the *concrete* set corresponding to a type, e.g. $\mathfrak{I}(\mathtt{int}) = \mathbb{Z}$, whereas a constraint is satisfied either if the variable

ranges in the correct set:

$$\mathbb{R}[\![\mathtt{x} = \mathtt{t}]\!](v) \Leftrightarrow v \in \mathcal{I}(t)$$

or the two variables range over the same set:

$$\mathbb{R}[\![\mathtt{x}_1 = \mathtt{x}_2]\!](v_1, v_2) \Leftrightarrow \exists t.\ v_1, v_2 \in \mathcal{I}(t).$$

Eventually dropping eliminates all the relations involving a given variable and $\curlyvee$ is the most general unifier (mgu) operation [66].

It can be shown that the construction fulfills the requirements of the last section so it can be used to analyze a module. But why doing it? It is good for example to perform a modular analysis of monomorphic types. In fact a module can be analyzed with polymorphic types and then instantiated to obtain a monotype. For example the function `hd`, that returns the first element in a list, has polytype α `list` $\rightarrow \alpha$ so that for `hd`$([2,3,5,8])$ we can simply instantiate α with `int` to obtain its monotype `int`.

6.4.2 Relevant Context Inference

A further example is Relevant Context Inference (RCI) whose main idea is to abstract module semantics by relations between the module caller context and the result. [18] introduced an application of RCI to a points-to analysis for the C++ language. Roughly, the analysis assigns symbolic names to input, and global, values then propagated inside the method body. The result is finally obtained by dropping the relations containing local variables.

We claim that this analysis can be easily encapsulated in our framework. In fact the *"dataflow elements (dfelms)"* of the analysis are terms in the form $\langle rc,\ points\text{-}to\rangle$ where rc, the relevant context is a condition on the possible aliasing and/or the type of the module input values and *points-to* is a pair specifying which variable points to which object. The meaning is that if rc holds then so *points-to*. Assignment at a program point $\mathtt{pp_i}$ is modeled according to Lemma 6.7 by setting

$$\begin{aligned}\rho(\mathtt{x{:=}E}) = \{\langle\emptyset, \langle \mathtt{x}_{(\mathtt{pp_{i+1}})}, \mathtt{E}\rangle\} \cup \{\langle\emptyset, \langle \mathtt{y}_{(\mathtt{pp_i})},\ \mathtt{y}_{(\mathtt{pp_{i+1}})}\rangle\rangle \mid \mathtt{y} \neq \mathtt{x}\} \\ \cup \{\langle rc, pt\rangle \mid \forall \mathtt{x}'_{(\mathtt{in})} \in \mathtt{PP_{in}}.\ (rc = \langle \mathtt{x}\ \mathbf{eq}\ \mathtt{x}'_{(\mathtt{in})}\rangle,\ pt = \langle \mathtt{x}'_{\mathtt{pp}_{(\mathtt{i+1})}},\ \mathtt{E}\rangle) \\ \vee\ (rc = \langle \mathtt{x}\ \mathbf{neq}\ \mathtt{x}'_{(\mathtt{in})}\rangle, pt = \langle \mathtt{x}'_{(\mathtt{pp_{i+1}})},\ \mathtt{x}'_{\mathtt{pp}_{(\mathtt{i})}}\rangle)\}.\end{aligned}$$

The intuitive meaning is that after an assignment the variable `x` has the value of the right operand, all the others maintain the value they had at $\mathtt{pp_i}$ and all the possible aliasings with the input must be considered. Assignment semantics is obtained by keeping the variables at $\mathtt{pp_{i+1}}$ and $\mathtt{pp_{in}}$, so referring

to Lemma 6.7 we drop all the variables but that in $V = \mathtt{PP_{in}} \cup \mathtt{PP_{i+1}}$. Eventually the $\delta_{\mathtt{y}}$ consists in the transitive closure followed by elimination of such *dfelms* containing y. Analysis soundness immediately follows from Theorem 6.1.

The formulation of RCI as $\mathcal{A}$-domain is twofolds: first the soundness proof comes out from the previous section theory and second the analysis is formalized and not only explained using examples.

6.4.3 Incremental Modular Analysis

Last example is about the combination of analyses. The basic idea is to refine the result of a relational analysis by approximating the value of local variables whenever the first is not precise enough. To the best of our knowledge this is the first time that it used for modular analysis. In modular interval analysis $\mathtt{c}[\mathtt{x_{in}}, \mathtt{x_{out}}, \mathtt{x_{loc}}]$ is a set of linear equations and $\mathtt{x_{loc}}$ can be approximated by performing a worst-case analysis on a different domain. For example this C function:

```
int f (int x) {
  int b,c;
  c = getint();
  if (isEven(2 * c))
    b = 2*x+1;
  else
    b = 2*x;
  return (y = b − 2*x);
}
```

when analyzed with the polyhedra domain returns the following system of linear equations:

$$\mathtt{c}[\mathtt{x}, \mathtt{y}, \mathtt{b}] = \{0 \leq \mathtt{y} \leq 1,\ \mathtt{y} + 2 \cdot \mathtt{x} - \mathtt{b} = 0\}.$$

If we drop internal information we get

$$\delta_{\mathtt{b}}(\mathtt{c}[\mathtt{x}, \mathtt{y}, \mathtt{b}]) = \{0 \leq \mathtt{y} \leq 1\}$$

that is a rough approximation, since even the relation with the input value is lost. However, if we analyze the the code on the parity domain [29] with $\mathtt{x} = \top$, i.e. assuming the worst-case we obtain that $\mathtt{b} = \mathtt{odd}$. Then this result in conjunction with $\mathtt{c}[\mathtt{x}, \mathtt{y}, \mathtt{b}]$ shows that if $\mathtt{y} = 0$ then the system is infeasible. Therefore $\mathtt{y} = 1$ for all the possible inputs.

Our conjecture is that the method illustrated in the previous example is an approximation of the reduced product [31] of the two domains. However

it is effective in that the operations do not need to be redefined and it is faster since the second analysis is performed only if the result of the first is not precise enough.

6.5 Discussion

We presented a generic abstract domain for symbolic modular analysis and we proved its soundness w.r.t. a domain of traces. We showed that the construction is general enough to cope with some existing modular analyses and we give a hint on how the results can be improved by combining two kinds of modular analyses, the symbolic and the worst-case one.

In the future, we plan to go deeper in investigating to refine the result of a constraint based modular analysis by partitioning traces. Intuitively, abstraction is performed not on a module seen as a big set of traces, but on a partition of this set, so that the analysis results might be better.

Chapter 7

Symbolic Relations for Approximating the Class Semantics

> We all agree that your theory is crazy, but is it crazy enough?
>
> Niels Bohr (1926)

In this chapter we study the particular case when the semantics of the methods is approximated by symbolic relations, i.e. the methods are analyzed using an instance of the $\mathcal{A}$-domain. In particular, we show how starting from a class `A` we can derive an approximated class $\bar{\texttt{A}}$ to be used either as a class documentation or as a tester for a client using `A`. Furthermore, $\bar{\texttt{A}}$ can be used for the inference of two kinds of class invariants, enabling fine tuning of the cost/precision ratio.

This chapter is based on the published work [74].

7.1 Introduction

Exploiting the results of the previous chapter, the class constructor and methods input/output behavior can be approximated by a set of constraints *symbolically* relating input values (instance fields and formal parameters) with outputs (updated instance fields and return value). Therefore, given a class `A` each one of its methods `m` is approximated by a set of constraints $\mathtt{c_m}[\mathtt{x_{in}}, \mathtt{x_F}, \mathtt{x_{out}}, \mathtt{x_{F'}}]$ where $\mathtt{x_{in}}$ and $\mathtt{x_F}$ are respectively the method formal parameters and the class fields at `m`'s entry-point, $\mathtt{x_{out}}$ is the return value (if any) and $\mathtt{x_{F'}}$ symbolically represent the values of class fields at the exit-point.

An immediate application is the automatic generation of documentation: as the constraints describe the methods' behavior, they can be shipped as class documentation. The main advantage is that they are directly generated from the source and no human intervention is required when the code changes: it is sufficient to (re-)run the analysis on the modified methods. Another application is the derivation of an abstract[1] class $\bar{\texttt{A}}$ to be used in full program analysis: the methods of `A` are replaced by their abstract counterparts $\texttt{c}_\texttt{m}$.

In whole-program analysis the use of an (already computed) abstraction of `m` brings to a global analysis speedup. From such a point of view, our approach is close to [95]. In such a work the authors address the problem of verifying whether the client of a class uses it in a *correct* way. To do it, they essentially derive a symbolic class invariant and they check that a given client does not violate it. However, our work is different since we do not require any human interaction for deriving the methods' approximations and our analysis does not necessarily need a client in order to derive class integrity properties.

A further use of $\bar{\texttt{A}}$ is for the inference of class invariants. In fact as the method semantics is approximated once and for all, the fixpoint computation of (5.8) can be approximated so to reduce its cost. In particular, we exploit $\bar{\texttt{A}}$ in order to obtain two kinds of class invariants $I_\texttt{A}$ and $J_\texttt{A}$ the first being cheap to compute but imprecise and the second one being more expensive but more precise, too.

7.2 An Example of Stack

We will illustrate the results of this chapter on the Java class, taken from [67], in Figure 7.1. The class `Stack` implements a stack parameterized by its dimension, specified at object creation time. The annotations on the right have been automatically derived by instantiating the results of this chapter. The comments at lines `6` to `8` specify the class invariant, i.e. a property that, for each possible instantiation of the class `Stack`, is valid before and after the execution of each method in the class. In particular, it states that the size of the stack is always greater than zero and it does not change during the execution. Moreover the stack pointer `pos` is always positive and smaller or equal to the array size.

The inferred class invariant can be used either to automatically produce

[1] In this context we employ the term *abstract* in the sense of abstract interpretation theory and not of object-oriented programming: hence by an abstract class $\bar{\texttt{A}}$ we mean a *semantic object* that approximates the semantics of `A`, and not a class that should not be instantiated [56].

```
class StackError extends Exception {
}

public class Stack {

  private int size;              // 1 <= size
  private int pos;               // 0 <= pos <= size
  private Object[] stack;        // size = stack.Length

  Stack(int size) {
    this.size = Math.max(size,1);
    this.pos = 0;
    this.stack = new Object[this.size];
  }

  boolean isEmpty() {
    return (pos <= 0);
  }

  boolean isFull() {
    return (pos >= size);
  }

  Object top() {
    return stack[pos-1];         // -1 <= pos-1 < stack.Length
  }

  void push(Object o) throws StackError {
    if(!isFull()) {
      stack[pos] = o;            // 0 <= pos < size
      pos++;
    } else
    throw new StackError();
  }

  void pop() throws StackError {
    if(!isEmpty())
      pos--;
    else
      throw new StackError();
  }
}
```

Figure 7.1: Java source code and annotations for the `Stack` class

the code documentation or to point out possible runtime errors as for example at line 25 where a negative array access may be performed if the stack is empty. Moreover the same property can be used to optimize the generated bytecode: in fact for each possible instantiation of the `Stack` and for each possible calling context it is sure that the array upper bound is never accessed or overcome so that this check can be avoided in the compiled code. Analogously at line 30 at bytecode level the array checks can be omitted as it is proved that `pos` is *always* in the array boundaries.

7.3 First Abstraction: Approximating Classes

In this section we show how to build a class approximation on the top of methods abstraction. The idea is quite simple: given a class `A` the *abstract* class $\bar{\texttt{A}}$ can be obtained as follows:

- the constructor `init` is replaced with an a symbolic relational approximation of its semantics $\mathsf{c}_{\texttt{init}}$;
- each one of the methods `m` is replaced with the corresponding symbolic relational approximation $\mathsf{c}_{\texttt{m}}$;
- each field `f` of type `T` is replaced with a field $\bar{\texttt{f}}$ of type $\bar{\texttt{T}}$.

7.3.1 Definition of an Abstract Class

We need some preliminary definitions, before giving the formal definition of abstract classes. First we define the subset $\text{initial}_{\mathsf{C}}$ of the constraints that are about just the initial states:

Definition 7.1 (Initial Constraints, $\text{initial}_{\mathsf{C}}$) *Let* `in` *be the entry point of the constructor or of a method. Then, the subset of initial constraints* $\text{initial}_{\mathsf{C}} \in \mathcal{P}(\mathsf{C})$ *is defined as follows:*

$$\text{initial}_{\mathsf{C}} = \{\mathsf{c} \mid \forall \mathtt{x}_{(\mathtt{pp})} \in \text{tiedVars}(\mathsf{c}).\ \mathtt{pp} = \mathtt{in}\}.$$

It is worth noting that if $\mathsf{c}_0 \in \text{initial}_{\mathsf{C}}$, then $\gamma_{\mathsf{C}}(\mathsf{c}_0)$ is a set of traces of length one, i.e. of states.

Next we define the relation $\Subset$, which intuitively states that a constraint is a sound approximation of the method collecting semantics. Roughly speaking, the definition below says that if the input states of a method are soundly approximated by an initial constraint c_0, then $\mathsf{c} \curlywedge \mathsf{c}_0$ is a sound approximation of the method collecting semantics.

Definition 7.2 (Approximation of a Method, $\Subset$) *Let* $\mathtt{m} \in \mathtt{M}$, $\mathtt{c} \in \mathsf{C}$ *and* $\alpha_{\dashv} \in [\mathcal{T}(\Sigma) \to \mathcal{P}(\Sigma)]$ *be the abstraction defined in the Example 2.1. Then the relation* $\Subset$ *is defined as follows:*

$$\mathtt{m} \Subset \mathtt{c} \iff \forall S \in \mathcal{P}(\Sigma).\ \forall \mathtt{c}_0 \in \text{initial}_{\mathsf{C}}. \\ S \subseteq \gamma_{\mathsf{C}}(\mathtt{c}_0) \Longrightarrow \mathbb{M}[\![\mathtt{m}]\!](S) \subseteq \alpha_{\dashv} \circ \gamma_{\mathsf{C}}(\mathtt{c} \curlywedge \mathtt{c}_0).$$

In such a case, we say that c *approximates* m*.*

The soundness requirement of the definition above implies that if $\mathtt{m} \Subset \mathtt{c}$ then, in particular, c contains the information about the fields that may escape from the method m. We denote such fields with $\mathcal{E}(\mathtt{c})$.

The definition of $\Subset$ can be easily extended to cope with the approximation of the semantics of the constructor, so that we have all the elements for the definition of the abstract class:

Definition 7.3 (Abstract Class, $\bar{\mathtt{A}}$) *Let* $\mathtt{A} = \langle \mathtt{init}, \mathtt{F}, \mathtt{M} \rangle$ *be a class. An abstract class* $\bar{\mathtt{A}}$ *is a triplet* $\langle \mathtt{c}_{\mathtt{init}}, \{\bar{\mathtt{T}}_i\ \bar{\mathtt{f}}_i\}, \{\mathtt{c}_{\mathtt{m}}\} \rangle$ *where:*

- $\mathtt{c}_{\mathtt{init}}$ *approximates the constructor:* $\mathtt{init} \Subset \mathtt{c}_{\mathtt{init}}$*;*
- *each field* $\bar{\mathtt{f}}_i$ *of type* $\bar{\mathtt{T}}_i$ *replaces the corresponding field* $\mathtt{f}_i \in \mathtt{F}$ *of type* $\mathtt{T}_i$*;*
- *each* $\mathtt{c}_{\mathtt{m}}$ *approximates the corresponding method* $\mathtt{m} \in \mathtt{M}$*:* $\mathtt{m} \Subset \mathtt{c}_{\mathtt{m}}$*.*

In the following, with an abuse of notation, we will write $\bar{\mathtt{F}}$ for the abstract fields and $\bar{\mathtt{M}}$ for the set of constraints approximating the semantics of methods.

7.3.2 Applications

The approximation $\bar{\mathtt{A}}$ has two immediate applications. The first one is as documentation. In fact, constraints describe the methods' behavior, so that they can be used as a description of the compiled code. The main advantage of this way of doing is that the documentation is obtained automatically from the source code and not from user annotations [102], saving programmer time and being less error-prone.

The second one is for the abstract debugging of a client of the class. In fact suppose to have a program P using the class A. In a static analysis of P we can employ $\bar{\mathtt{A}}$ either to save time by avoiding the analysis of A's methods at each (abstract) invocation or to test that P uses the class in a correct way, e.g. in the Stack example it never pops an element from an empty stack. On the other hand, a similar form of abstract debugging, in which the semantics of an invocation is replaced by a *summary* function, is used in ESC/Java to

perform modular analysis of classes [47], in [95] to perform modular analysis of Java containers and in [94] to perform modular verification of concurrent Java programs.

EXAMPLE 7.1 (ABSTRACT STACK, $\overline{\texttt{Stack}}$) With reference to our running example, the constructor and the methods of `Stack` can be, automatically, approximated by set of linear inequalities in order to obtain the abstract class $\overline{\texttt{Stack}}$.

We begin with the approximation of fields. As `size` and `pos` are of integer type, we leave them unchanged. On the other hand, `stack` is an array object that we chose to approximate with its length, i.e. we abstract away the values and the representation of array by just keeping its length. Therefore $\overline{\texttt{Object[]}} = \texttt{int}$ and

$$\bar{\texttt{F}}_{\texttt{Stack}} = \{\texttt{int size}, \texttt{int pos}, \texttt{int stacklen}\}.$$

The approximation of the constructor as well as that of methods is obtained by giving formal names (e.g. $\texttt{pos}_{\texttt{F}}$) to the values of the actual parameters and instance fields corresponding to the initial values at method entry-point and by establishing a relation with the final value (e.g. $\texttt{pos}_{\texttt{F}'}$) of these variables. In our case this relation can be established by abstractly executing the methods on the polyhedra abstract domain refined with trace partitioning [59].

Therefore, if `in` and `out` are the program points corresponding respectively to the entry-point and exit-point of the constructor, then the approximation of the class constructor is:

$$\begin{aligned}\texttt{c}_{\texttt{Stack()}} = \{&\texttt{size}_{\texttt{(in)}} \leq \texttt{size}_{\texttt{(out)}},\ 1 \leq \texttt{size}_{\texttt{(out)}},\\ &\texttt{stacklen}_{\texttt{(out)}} = \texttt{size}_{\texttt{(out)}},\ \texttt{pos}_{\texttt{(out)}} = 0\}.\end{aligned}$$

Roughly, it states that the initial size of the stack can never be smaller than 1 or smaller than the value of the constructor parameter. Furthermore, the length of the array is equal to the value of the instance field `size` and the stack pointer `pos` is initially set to 0.

The approximation for the helper methods `isEmpty` and `isFull` is below. The return value is, without any loss of generality, assumed to be stored in the variable $\texttt{x}_{\texttt{(out)}}$. `F` and `F'` are respectively the field values before and after the execution of a method.

$$\begin{aligned}\texttt{c}_{\texttt{isEmpty}} &= \{\texttt{x}_{\texttt{(out)}} = (\texttt{pos} \leq 0)\} \cup \{\texttt{F} = \texttt{F}'\}\\ \texttt{c}_{\texttt{isFull}} &= \{\texttt{x}_{\texttt{(out)}} = (\texttt{pos} \geq \texttt{size})\} \cup \{\texttt{F} = \texttt{F}'\}.\end{aligned}$$

Such an approximation states that an invocation of the two methods does not change the values of the object fields.

The approximation of `top` must take into account the array access. In fact, if the pointer `pos` is out of the bounds then the exception `ArrayIndexOutOfBoundsException` is thrown. We assume that $\Omega(\texttt{Exc})$ denotes that the exception `Exc` is thrown.

$$\begin{aligned}\mathtt{c}_{\mathtt{top}} = \{&\text{if } (\mathtt{pos}_{(\mathtt{in})} > \mathtt{size}_{(\mathtt{in})}) \text{ or } (\mathtt{pos}_{\mathtt{F}} \leq 0)\\ &\quad \text{then } \mathtt{x}_{\mathtt{out}} = \Omega(\mathtt{ArrayIndexOutOfBoundsException})\\ &\quad \text{else } \mathtt{F} = \mathtt{F}'\}\end{aligned}$$

It is worth noting that in $\mathtt{c}_{\mathtt{top}}$ if no exception is thrown then the value returned is abstracted to be everything.

Finally, we are left with the approximations of `push` and of `pop`, given below. $\mathtt{F}_{\mathtt{p}}$ and $\mathtt{F}'_{\mathtt{p}}$ are as above except that they do not contain, respectively, the variables $\mathtt{pos}_{(\mathtt{in})}$ and $\mathtt{pos}_{\mathtt{out}}$:

$$\begin{aligned}\mathtt{c}_{\mathtt{push}} = \{&\text{if } (\mathtt{pos}_{(\mathtt{in})} < \mathtt{size}_{(\mathtt{in})})\\ &\quad \text{then if } (\mathtt{pos}_{(\mathtt{in})} \geq 0)\\ &\quad\quad \text{then } \mathtt{pos}_{(\mathtt{out})} = \mathtt{pos}_{(\mathtt{in})} + 1, \mathtt{pos}_{(\mathtt{out})} > 0, \mathtt{pos}_{(\mathtt{out})} \leq \mathtt{size}_{(\mathtt{in})}\\ &\quad\quad \text{else } \mathtt{x}_{(\mathtt{out})} = \Omega(\mathtt{ArrayIndexOutOfBoundsException})\\ &\quad \text{else } \mathtt{x}_{(\mathtt{out})} = \Omega(\mathtt{StackError})\} \cup \{\mathtt{F}_{\mathtt{p}} = \mathtt{F}'_{\mathtt{p}}\}\\ \mathtt{c}_{\mathtt{pop}} = \{&\text{if } (\mathtt{pos}_{(\mathtt{in})}) > 0)\\ &\quad \text{then } \mathtt{pos}_{(\mathtt{out})} = \mathtt{pos}_{(\mathtt{in})} - 1, \mathtt{pos}_{(\mathtt{out})} \geq 0\\ &\quad \text{else } \mathtt{x}_{(\mathtt{out})} = \Omega(\mathtt{StackError})\} \cup \{\mathtt{F}_{\mathtt{p}} = \mathtt{F}'_{\mathtt{p}}\}.\end{aligned}$$

Finally, the abstract class $\overline{\texttt{Stack}}$ is the triplet

$$\langle \mathtt{c}_{\mathtt{Stack()}}, \bar{\mathtt{F}}_{\mathtt{Stack}}, \{\mathtt{c}_{\mathtt{isEmpty}}, \mathtt{c}_{\mathtt{isFull}}, \mathtt{c}_{\mathtt{top}}, \mathtt{c}_{\mathtt{push}}, \mathtt{c}_{\mathtt{pop}}\}\rangle.$$

From the last example it follows that if a class `A` uses a class `B`, e.g. it has a field of type `B`, then either the abstract class $\bar{\mathtt{B}}$ must be available before the derivation of the abstract class $\bar{\mathtt{A}}$ or, if the two are mutually dependent, the derivation of $\bar{\mathtt{A}}$ and $\bar{\mathtt{B}}$ must be performed at the same time.

7.3.3 Checking the Well-behavior of a Client

An interesting use of $\overline{\texttt{Stack}}$ is to check for client well-behavior, i.e. to check that a client which uses this class does not cause an exception raised by pushing an object on a full stack, or calling `top` on the empty stack. Therefore we can say that $\overline{\texttt{Stack}}$ abstracts away from `Stack` the effective values of the array but it behaves in the same way with respect to the *exceptional* behavior.

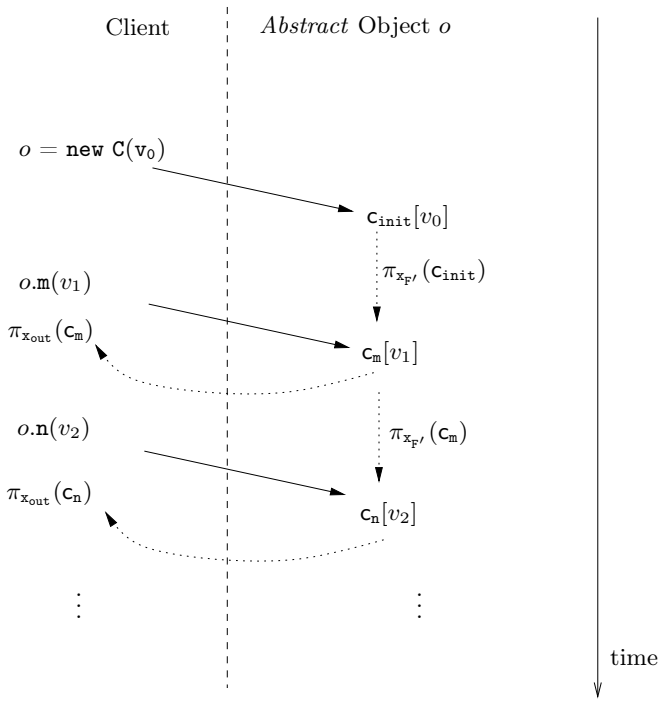


Figure 7.2: A schema of a client using an instance of an abstract class $\bar{\mathtt{A}}$

In Figure 7.2 a general schema is given for performing a client debugging: at first the client instantiates a new object $\mathtt{o}$ of the abstract class $\bar{\mathtt{A}}$. This gives out an approximation of the internal object fields $\mathtt{o.x_{F'}}$. In our running example it essentially reduces to give the stack size. Then, whenever the client calls a method $\mathtt{m}$ with an input value v, the corresponding approximation $\mathtt{c_m}[\mathtt{x_{(in)}}, \mathtt{x_F}, \mathtt{x_{(out)}}, \mathtt{x_{F'}}]$ is fetched and instantiated with $\mathtt{x_{(in)}} = v$ and $\mathtt{x_F} = \mathtt{o.x_{F'}}$ resulting in the new set of constraints:

$$\mathtt{c'}[\mathtt{x_{(out)}}, \mathtt{x_{F'}}] = \mathtt{c_m}[\mathtt{x_{(in)}}, \mathtt{x_F}, \mathtt{x_{(out)}}, \mathtt{x_{F'}}] \curlywedge \{\mathtt{x_{(in)}} = v, \mathtt{x_F} = \mathtt{o.x_{F'}}\}.$$

Eventually the return value for the client is obtained by keeping the value of $\mathtt{x_{out}}$, hence $\pi_{\mathtt{x_{out}}}(\mathtt{c'}[\mathtt{x_{out}}, \mathtt{x_{F'}}])$. Analogously the new $\mathtt{o}$ internal state is $\mathtt{o.x_{F'}} = \pi_{\mathtt{x_{F'}}}(\mathtt{c'}[\mathtt{x_{out}}, \mathtt{x_{F'}}])$.

Example 7.2 (Stack Underflow) Let us suppose to have a context that creates an instance of the class `Stack` as the one given below:

```
int i;
s = new Stack(10);

s.push(new Integer(123));
```

```
5 i = s.pop();
  s.top();
```

In such a case we have that just after the invocation of the constructor the object internal state is approximated by the following constraints:

$$\mathtt{c}_0 = \{\mathtt{size}_{(\mathtt{out})} = \mathtt{stacklen}_{(\mathtt{out})} = 10, \mathtt{pos}_{(\mathtt{out})} = 0\}.$$

The internal state `s` after the invocation of `push` is:

$$\begin{aligned}
&\pi_{\mathtt{x}_{\mathtt{F}'}}(\mathtt{c}_{\mathtt{push}}[\mathtt{size}_{(\mathtt{in})}, \mathtt{pos}_{(\mathtt{in})}, \mathtt{x}_{\mathtt{out}}, \mathtt{size}_{\mathtt{F}'}, \mathtt{pos}_{\mathtt{F}'}] \curlywedge \mathtt{c}_0)\\
=&\pi_{\mathtt{x}_{\mathtt{F}'}}(\{\mathtt{size}_{(\mathtt{out})} = \mathtt{stacklen}_{(\mathtt{out})} = 10, \mathtt{pos}_{(\mathtt{out})} = 1,\\
&\quad \mathtt{pos}_{(\mathtt{out})} > 0, \mathtt{pos}_{(\mathtt{out})} \leq 10\})\\
=&\{\mathtt{size}_{(\mathtt{out})} = \mathtt{stacklen}_{(\mathtt{out})} = 10, \mathtt{pos}_{(\mathtt{out})} = 1\}.
\end{aligned}$$

Invoking `pop` in such a state causes the pointer to be reset to 0, so that the object internal state becomes:

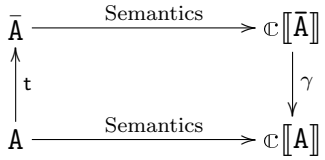
$$\mathtt{c}_2 = \{\mathtt{size}_{(\mathtt{out})} = \mathtt{stacklen}_{(\mathtt{out})} = 10, \mathtt{pos}_{(\mathtt{out})} = 0\}.$$

Finally, the invocation of `top` causes the raising of the runtime exception:

$$\begin{aligned}
&\pi_{\mathtt{x}_{(\mathtt{out})}}(\mathtt{c}_{\mathtt{top}}[\mathtt{size}_{(\mathtt{in})}, \mathtt{pos}_{(\mathtt{in})}, \mathtt{x}_{\mathtt{out}}, \mathtt{size}_{(\mathtt{out})}, \mathtt{pos}_{(\mathtt{out})}] \curlywedge \mathtt{c}_2)\\
=&\pi_{\mathtt{x}_{(\mathtt{out})}}(\{\mathtt{x}_{(\mathtt{out})} = \Omega(\mathtt{ArrayIndexOutOfBoundsException})\}\\
=&\Omega(\mathtt{ArrayIndexOutOfBoundsException}).
\end{aligned}$$

7.3.4 Soundness

The soundness comes out from the following observation: the construction proposed in Definition 7.3 is a program transformation, i.e. a meaning-preserving mapping defined on the program syntax [36]. In fact, the above definition is equivalent to a program transformer `t` on the syntax and a concretization function γ such that the following diagram commutes:



Differently stated, if we transform the class source and we take its semantics then we obtain an upper approximation of the original class semantics. Formally, the following theorem holds:

THEOREM 7.1 (SOUNDNESS OF THE ABSTRACT CLASS) *The abstract class* $\bar{\mathtt{A}}$ *is a sound approximation of the semantics of* $\mathtt{A}$.

Proof. (Sketch) The semantics of an abstract class may be defined in the same way as the trace semantics of classes introduced in Chapter 3. In such a case a transition is an instance of a method approximation.

By definition, the constructor and the methods of $\bar{\mathtt{A}}$ are such that $\mathtt{c}_{\mathtt{m}} \Subset \mathtt{m}$. As a consequence, each transition caused by a method $\mathtt{m}$ is upper-approximated by a transition caused by the corresponding approximation $\mathtt{c}_{\mathtt{m}}$. Finally, the thesis follows from the fixpoint transfer theorem (cf. Theorem 2.6) and the fact that the fields and the classes they belong to are finitely many.

q.e.d.

7.4 Second Abstraction: Class Invariants

On the base of the abstract classes previously introduced, in this section we show how to automatically derive a class invariant.

7.4.1 History-insensitive Class Invariant

If $\mathtt{A}$ has, for example, just two methods $\mathtt{m}_1$ and $\mathtt{m}_2$, respectively approximated by $\mathtt{c}_{\mathtt{m}_1}$ and $\mathtt{c}_{\mathtt{m}_2}$, then the property "$\mathtt{c}_{\mathtt{m}_1}$ or $\mathtt{c}_{\mathtt{m}_2}$" is always true for each instance of $\mathtt{A}$ and for each calling context. Thus "$\mathtt{c}_{\mathtt{m}_1}$ or $\mathtt{c}_{\mathtt{m}_2}$" is a class invariant. Formally:

$$I_{\mathtt{A}} = \pi_{\mathtt{x}_{\mathtt{F}'}}(\mathtt{c}_{\mathtt{m}_1}) \curlyvee \pi_{\mathtt{x}_{\mathtt{F}'}}(\mathtt{c}_{\mathtt{m}_2}).$$

In general a class invariant can be obtained by gathering all the method approximations of the object local environment, so that we can state the following theorem:

THEOREM 7.2 (HISTORY-INSENSITIVE CLASS INVARIANT) *Let* $\mathtt{A} = \langle \mathtt{init}, \mathtt{F}, \mathtt{M} \rangle$ *be a class and* $\bar{\mathtt{A}} = \langle \mathtt{c}_{\mathtt{init}}, \bar{\mathtt{F}}, \bar{\mathtt{M}} \rangle$ *the corresponding abstract class. Furthermore, let*

$$\mathtt{c}_{\mathtt{A}} = \mathtt{c}_{\mathtt{init}} \curlyvee \bigvee_{\mathtt{m} \in \bar{\mathtt{A}}} \pi_{\mathtt{x}_{\mathtt{F}'}}(\mathtt{c}_{\mathtt{m}}) \,.$$

Then

$$I_{\mathtt{A}} = \delta_{\mathcal{E}(\mathtt{c}_{\mathtt{A}})}(\mathtt{c}_{\mathtt{A}}) \tag{7.1}$$

is a class invariant for $\mathtt{A}$. *Moreover each* $I'_{\mathtt{A}}$ *such that* $I_{\mathtt{A}} \preceq I'_{\mathtt{A}}$ *is a class invariant.*

Proof. Let us consider the formulation of the strongest class invariant of (5.2). By theorem hypotheses, the constraints approximate both the initial states and the collecting semantics of the methods for all possible inputs. Thus the fixpoint computation of (5.8) terminates after one step. The soundness follows from definition of join Υ, from the property *(extensivity)* of π, and from the fact that we drop the fields that may escape from the object scope (i.e. $\mathcal{E}(\mathtt{c_A})$).

q.e.d.

We call (7.1) a history-insensitive class invariant since the method invocation history is not considered in method's analyses, as can be seen in the following example:

Example 7.3 (I-Class Invariant for `Stack`) If we apply (7.1) to the `Stack` example, we obtain:

$$\begin{aligned} I_{\mathtt{Stack}} \preceq\ & \{\mathtt{pos}_{\mathtt{F}'} = 0\} \Upsilon \{0 < \mathtt{pos}_{\mathtt{F}'} \leq \mathtt{size}_{\mathtt{F}'}\} \Upsilon \{0 \leq \mathtt{pos}_{\mathtt{F}'}\} \\ =\ & \{0 \leq \mathtt{pos}_{\mathtt{F}'}\} \end{aligned}$$

since `isEmpty`(), `isFull`() and `top`() do not modify the object fields. The $\preceq$ originates from the fact that, for example purposes, we do not consider the value of `stack.length` and that `size` is not modified after constructor invocation. The intuitive meaning of $I_{\mathtt{Stack}}$ is that whatever method is invoked the field variable `pos` is positive.

The invariant discovered in the last example immediately allows us to point out that at line `25` of Figure 7.1 there is a possible out of the bounds array access. The programmer did not provide any check for it, so that if the stack is empty (i.e. `pos` $= 0$) a runtime exception is thrown. However $I_{\mathtt{Stack}}$ does not give any upper bound for the stack pointer value. Hence the code generated for lines `25` and `30` cannot be optimized by removing the check `pos` $<$ `stack.length`. In the next section we show how to obtain a more precise (but more expensive to compute) class invariant.

7.4.2 History-sensitive Class Invariant

When carefully looking at (7.1) it is possible to realize that the result can be improved by exploiting the inter-relations between different methods. Differently stated, proceeding as in last section, we completely abstract away from the method invocation history. In particular (7.1) does not consider the fact that the first called method is the class constructor, that intuitively sets up

the object local environment. So, in order to avoid this problem we propose an iterative algorithm to compute a class invariant.

The first step performs an approximation of the object internal state after the invocation of its constructor, i.e. $J^0 = \pi_{\mathtt{x}_{\mathtt{F}'}}(\mathtt{c}_{\mathtt{init}})$. After that an approximation of the method's semantics is computed with the object internal state specified by J^0, obtaining for each method $\mathtt{m}$ a

$$J^1_{\mathtt{m}} = \mathtt{c}_{\mathtt{m}} \curlywedge J^0.$$

Intuitively this can be justified as follows: after the object creation, the client invokes the method $\mathtt{m}_{\mathtt{i}}$. Thus, when the method returns it has changed the object internal state, that is now $\pi_{\mathtt{x}_{\mathtt{F}'}}(J^1_{\mathtt{m}})$. Now, since we are performing a modular analysis, i.e. without considering the client context, we do not know which method is called. Equivalently, we can say that the invoked method is either the first one, or the second one, or the third, etc. This writes down as:

$$\bigvee_{\mathtt{m}\in\mathtt{A}} \pi_{\mathtt{x}_{\mathtt{F}'}}(J^1_{\mathtt{m}}).$$

Eventually since we are interested in the class invariant, we must collect the effects (on the internal fields) of the first and the second step, obtaining:

$$J^1 = J^0 \curlyvee \left(\bigvee_{\mathtt{m}\in\mathtt{A}} \pi_{\mathtt{x}_{\mathtt{F}'}}(J^1_{\mathtt{m}}) \right).$$

Then we can iterate, obtaining $J^2, J^3, \ldots$. It is immediate to see that this sequence is an increasing chain: $J^0 \preceq J^1 \preceq J^2 \ldots \preceq J^k \preceq J^{k+1} \ldots$. Moreover, in general it is possible that it is infinite, i.e. the analysis does not terminate. So, as usual in abstract interpretation, in order to force the convergence of the sequence (and hence of the analysis) we need a widening operator [29]. We adopt the fixpoint strategy of (5.8) that consists in limiting the sequence of *exact* iterations to a given step n and then replacing the $\curlyvee$ with $\curlyvee$.

THEOREM 7.3 (HISTORY-SENSITIVE CLASS INVARIANT) *Let* $\bar{\mathtt{A}} = \langle \mathtt{c}_{\mathtt{init}}, \bar{\mathtt{F}}, \bar{\mathtt{M}} \rangle$ *be an abstract class for a class* $\mathtt{A}$ *and let* n *be an integer. Then the limit* $J_{\mathtt{A}}$ *of the sequence constructed as follows:*

$$\begin{aligned} J^0 &= \pi_{\mathtt{x}_{\mathtt{F}'}}(\mathtt{c}_{\mathtt{init}}) \\ J^{k+1} &= J^k \curlyvee \bigvee_{\mathtt{m}\in\bar{\mathtt{M}}} \pi_{\mathtt{x}_{\mathtt{F}'}}(\mathtt{c}_{\mathtt{m}} \curlywedge J^k) \curlyvee \delta_{\mathcal{E}(J^k)}(J^k) && \text{if } 0 \le k < n \\ J^{k+1} &= J^k \curlyvee \bigvee_{\mathtt{m}\in\bar{\mathtt{M}}} \pi_{\mathtt{x}_{\mathtt{F}'}}(\mathtt{c}_{\mathtt{m}} \curlywedge J^k)) \curlyvee \delta_{\mathcal{E}(J^k)}(J^k) && \text{if } k \ge n \end{aligned} \tag{7.2}$$

exists and it is a class invariant for $\mathtt{A}$.

Proof. The iteration schema above is an instance of (5.8). The convergence is assured by the definition of $\triangledown$. The soundness follows from the definition of $\bar{\mathtt{A}}$ and the properties of π, $\curlyvee$ and $\triangledown$. Note that escaping scope is handled by $\delta_{\mathcal{E}(J^k)}(J^k)$, which throws away the fields that escape from the object, so that it upper-approximates Context(·).

q.e.d.

The presented method is more precise but also more expensive as a fixpoint computation is introduced in addition to the local fixpoint computations for inferring the approximations $\mathtt{c}_{\mathtt{m}}$. Nevertheless the results are very good. In fact in the next example, 6 iterations are sufficient to compute $J_{\mathtt{Stack}}$, the stack invariant.

EXAMPLE 7.4 (J-CLASS INVARIANT FOR `Stack`) Applying the iteration strategy of (7.2) with $n = 4$, it is possible to discover a class invariant for the class `Stack`. We begin by considering the abstract class $\overline{\mathtt{Stack}}$ derived in the Example 7.1.

The first step corresponds to the approximation of the constructor:

$$J^0 = \mathtt{c}_{\mathtt{init}} = \{1 \leq \mathtt{size}_{(\mathtt{out})},\ \mathtt{pos}_{(\mathtt{out})} = 0,\ \mathtt{stacklen}_{(\mathtt{out})} = \mathtt{size}_{(\mathtt{out})}\}$$

For the successive iterations we restricts to the methods `push` and `pop` since they are the only two to modify the object internal state. So, we obtain

$$J^1_{\mathtt{push}} = \mathtt{c}_{\mathtt{push}} \curlywedge J^0 = \{1 \leq \mathtt{size}_{(\mathtt{out})}, \mathtt{pos}_{(\mathtt{out})} = 1\}$$
$$J^1_{\mathtt{pop}} = \mathtt{c}_{\mathtt{pop}} \curlywedge J^0 = \{\emptyset\}$$

so that, thanks to the property *(unit)* of dropping operator, we obtain the next step:

$$J^1 = J^0 \curlyvee J^1_{\mathtt{push}} \curlyvee J^1_{\mathtt{pop}} = \{1 \leq \mathtt{size}_{(\mathtt{out})}, 0 \leq \mathtt{pos}_{(\mathtt{out})} \leq 1\}.$$

Next iterate is:

$$J^2_{\mathtt{push}} = \mathtt{c}_{\mathtt{push}} \curlywedge J^1 = \{1 \leq \mathtt{size}_{(\mathtt{out})}, 1 \leq \mathtt{pos}_{(\mathtt{out})} \leq 2\}$$
$$J^2_{\mathtt{pop}} = \mathtt{c}_{\mathtt{pop}} \curlywedge J^1 = \{1 \leq \mathtt{size}_{(\mathtt{out})}, \mathtt{pos}_{(\mathtt{out})} = 0\}$$

yielding

$$J^2 = J^1 \curlyvee J^2_{\mathtt{push}} \curlyvee J^2_{\mathtt{pop}} = \{1 \leq \mathtt{size}_{(\mathtt{out})}, 0 \leq \mathtt{pos}_{(\mathtt{out})} \leq 2\}.$$

Then the fixpoint computation goes on with:

$$J^3_{\mathtt{push}} = \mathtt{c}_{\mathtt{push}} \curlywedge J^2 = \{1 \leq \mathtt{size}_{(\mathtt{out})}, 1 \leq \mathtt{pos}_{(\mathtt{out})} \leq 3, \mathtt{pos}_{(\mathtt{out})} \leq \mathtt{size}_{(\mathtt{out})}\}$$
$$J^3_{\mathtt{pop}} = \mathtt{c}_{\mathtt{pop}} \curlywedge J^2 = \{1 \leq \mathtt{size}_{(\mathtt{out})}, 0 \leq \mathtt{pos}_{(\mathtt{out})} \leq 1\}$$

so

$$J^3 = \{1 \leq \mathtt{size}_{(\mathtt{out})}, 0 \leq \mathtt{pos}_{(\mathtt{out})} \leq 3, \mathtt{pos}_{(\mathtt{out})} \leq \mathtt{size}_{(\mathtt{out})}\}$$

and

$$J^4 = \{1 \leq \mathtt{size}_{(\mathtt{out})}, 0 \leq \mathtt{pos}_{(\mathtt{out})} \leq 4, \mathtt{pos}_{(\mathtt{out})} \leq \mathtt{size}_{(\mathtt{out})}\}.$$

At this point we can apply a widening operator order to extrapolate the sequence limit. The intuition is that at step n we have a constraint in the form of $\mathtt{pos}_{(\mathtt{out})} \leq n$, so a way to stabilize the chain is by upper-approximating it with $\mathtt{pos}_{(\mathtt{out})} \leq +\infty$, i.e. by removing it from the constraint system. As a consequence, it the resulting set of constraints

$$J_{\mathtt{Stack}} = \{1 \leq \mathtt{size}_{(\mathtt{out})}, 0 \leq \mathtt{pos}_{(\mathtt{out})} \leq \mathtt{size}_{(\mathtt{out})}\}$$

can be easily verified to be a fixpoint.

In the particular example that we consider, no gain of precision is obtained by postponing the application of the widening operator or by considering a more precise widening. However, in practice it may be useful to have a more flexible handling of widening thresholds [9] or more precise widenings on polyhedra [7].

It is worth noting as with this second method we are able to discover the class invariant $\mathtt{pos} \leq \mathtt{size}$ and to produce the code annotations of Figure 7.1.

7.4.3 On comparing the $I_{\mathtt{A}}$ and $J_{\mathtt{A}}$ invariants

We have pointed out, and we illustrated with an example, that the history-sensitive class invariant is more precise than the history insensitive one. So why not always use the second one? The reason is the cost, both in time and memory.

In fact it is clear that in general, in order to compute $J_{\mathtt{A}}$ we need more iterations than for $I_{\mathtt{A}}$ and in certain cases this computation may be so expensive to make the analysis in practice infeasible. So, in our opinion, the choice between the two must be made by carefully looking at the precision/cost trade-off. We consider some heuristics for two typical scenarios:

The first scenario is during code development. At this stage a tool that helps the programmer to quickly find source(s) of (potential) bugs is welcome. However such a tool is required to run in a reasonable amount of time. Thus the use of an invariant computation based on the history-insensitive method makes sense. For example, a tool adopting such a strategy can indicate at early stages of program developing the missing check at line 25.

The second scenario is at code shipping time. Suppose that the code exiting from the development cycle is ready to be shipped. At this stage the

cost of a long but precise analysis can be afforded. This because a precise analysis is useful

- to prove that the code is correct, e.g. it contains no run-time errors, all the exception raised are caught, etc. and
- to optimize the compiled code, e.g. removing superfluous checks, statically resolving the virtual methods calls, etc.

Therefore in this second scenario it is reasonable to think of a tool computing a $J_{\mathtt{A}}$-style class invariant.

Nevertheless, both the methods are in general less precise than the generic one of Chapter 5 but also less expensive to compute.

7.5 Discussion

In this chapter we showed how to perform a class-level modular analysis for object oriented languages when the semantics of methods is approximated using a symbolic relational abstract domain. We presented two successive abstractions:

- the first one is a program transformation that given a class `A`, allows the construction of a transformed version $\bar{\mathtt{A}}$ to be used for client conformance testing or for class documentation;
- the second one is the automatic discovery of class invariant to be used either for verification or as support for an optimizing compiler.

Some empirical experience suggests that symbolic relations are worthwhile for the the analysis of classes in that, intuitively, precision is gained by keeping relations between the state of objects at creation-time and their evolution over the time. Furthermore, in next chapter we show how a $I_{\mathtt{A}}$ invariant is useful for the modular inference of subclass invariants.

Chapter 8

Class Invariants in Presence of Inheritance

> There's no sense in being precise when you don't even know what you're talking about.
>
> John von Neumann (1930)

In this chapter we extend our framework in order to cope with inheritance. In particular, we address the problem of inferring a class invariant for a class `S` = "`E extends B`", for some base class `B` and extension `E`. A direct approach, i.e. an approach based on the expansion of the subclassing relation [90], has several drawbacks: it may cause a quadratic code blow-up and it does not allow sharing of computations, e.g. the base class must be re-analyzed for each one of its descendant. Furthermore, in some circumstances, the source code of `B` may not be available, so that the analysis cannot be done at all.

In order to overcome such problems, we present an extension of our framework in which the invariant for `S` is inferred by referring only to the *code* of the extension `E` and the *invariant* of the base class `B`. Such an approach solves the above-mentioned problems as the base class needs to be analyzed just once, and the result of the analysis may be used many times. Furthermore, the compiled code of `B` may be shipped with the class invariant, enabling the analysis of code that relies on third-part libraries.

This chapter is based on the published work [77].

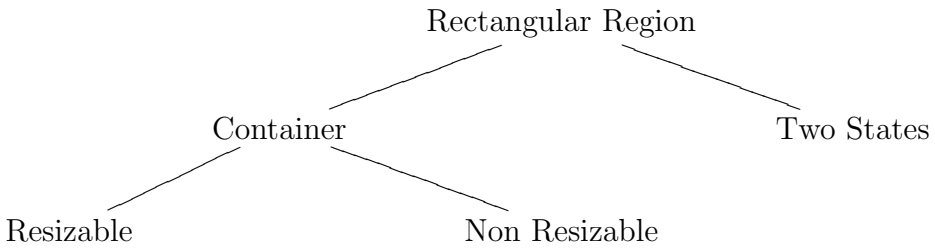


Figure 8.1: An example of conceptual hierarchy for a GUI

8.1 Inheritance

Classes do provide a good modular decomposition technique and possess many of the qualities expected of reusable components as they are homogeneous and coherent modules and interfaces can be clearly separated from their implementation according to the principle of information hiding. Nevertheless, more language support is needed in order to achieve the goals of code reusability and extendibility. Object-oriented languages provide such a support through the inheritance mechanism.

8.1.1 Inheritance in Software Development

Inheritance is a main concern in software development for two main reasons: in the design phase of a system, it enables the structuring of concepts in hierarchies and in the development phase, it enables code reuse and extension.

From a conceptual point of view, inheritance allows to model a system by factoring together the common structure of the entities that made up such a system. So, for instance, in a graphic toolkit resizable windows, dialog boxes and buttons all share some properties, e.g. they occupy a rectangular region on the screen and they must react to mouse actions. Furthermore resizable windows and dialog boxes may contain other windows, whereas buttons cannot. On the other hand, buttons are particular rectangular regions with just two visible states: pressed or not-pressed. Finally, resizable windows and dialog boxes are different in that the first may be able to change dynamically the area they occupy on the screen but not the latter. As a consequence, the common attributes can be smashed together so to obtain the conceptual hierarchy of Figure 8.1. For example, a dialog box is a non-resizable container. At its turn every container is a rectangular region, so that a dialog box is *even* a rectangular region.

From the programmer point of view, inheritance is a mechanism for in-

cremental programming. In particular, it allows to reach the two goals of reusability and extendibility. For reusability, it avoids to rewrite the same code over and over again, wasting time, introducing inconsistencies and risking errors. For extendibility, it allows to smartly extend existing code so to cope with new functionalities and needs. So, for instance in the above example the handling of the child windows is implemented by the class "Container". Such an implementation is shared by its subclasses "Resizable" and "Not Resizable".

8.1.2 Inheritance in Programming Languages

The notion of inheritance first appeared in Simula 67 [39]. In Simula, objects are grouped into classes and classes can be organized into a subclass hierarchy. Subclasses inherit all the attributes of their superclasses. Smalltalk [55] adopts and exploits the idea of inheritance, in particular by stressing the message-passing paradigm. Furthermore, with respect to Simula, Smalltalk abandons static scoping and strong typing in order to gain flexibility and to implement system introspection. Finally, Simula and Smalltalk both allow *multiple* inheritance, i.e. a class may have several incomparable superclasses.

Inheritance is also one of the main ingredients for the object-oriented extensions of procedural languages as C and Pascal. In particular the inheritance support in C++ [101] and Object Pascal [12] is very close to that of Smalltalk. On the other hand, in Objective-C [5] a class may *at most* have one superclass. Moreover, Objective-C introduces an orthogonal aspect to subclasses: categories. Roughly, categories allow to directly add methods to a class rather than define a subclass to extend it.

Last generation object-oriented languages as Java [56] and C# [84] provide a form of inheritance through class extension and interface implementation. Roughly, they have a weaker concept of multiple inheritance: a class has exactly one superclass but it may *implement* an arbitrary number of interfaces. An interface is a class without the implementation of the methods.

8.1.3 Semantics of Inheritance

In their seminal work on Simula [39], Dahl and Nygaard justified the concept of inheritance on syntactic bases, namely as textual concatenation of program blocks. A first semantic approach is [55] where the authors introduced an (informal) operational approach to the semantics of inheritance. In particular they reduced the problem of specifying the semantics of message dispatch to that of method lookup.

In the *objects as records model* [16], the semantics of an object is ab-

stracted with its type so that the inheritance identified with subtyping and the semantics of inheritance boils down to the subtyping relation. Nevertheless, such an approach is not fully satisfactory as shown in [22].

In [23] a denotational characterization of inheritance is introduced and proved correct w.r.t. to an operational semantics based on the method lookup algorithm of [55].

An unifying view of the different forms of inheritance provided by programming languages is presented in [13]. The authors present an inheritance mechanism, based on composition of mixins, that subsumes the others. Mixins are the CLOS [10] equivalents of Objective-C categories.

8.1.4 Inheritance and Class Invariants

In the Design by Contract (DbC) approach the inheritance is strictly tied to the preservation of class invariants, in that it enforces the preservation of the invariant by the subclass. For example, in Eiffel a subclass "inherits" the assertions of invariants of its superclasses. This means that if $A_{\texttt{B}}$ and $A_{\texttt{E}}$ are class assertions for respectively the base class `B` and the extension `E`, then the assertion of the subclass `S = E extends B` is "$A_{\texttt{E}}$ and $A_{\texttt{B}}$". Then, the Eiffel compiler generates stubs to check that at each entry point and exit point of methods the assertion "$A_{\texttt{E}}$ and $A_{\texttt{B}}$" is not violated.

The approach of Eiffel is a kind of dynamic behavioral subtyping [73, 4]. One drawback is that the inheritance relation is not decidable. In fact, as the assertion language is a side-effects free subset of Eiffel, in particular it is as expressive as Turing machines. Thus it is not possible to statically check that a subclass preserves the assertions of the superclass. We will cope with this aspect in the next chapter, where we present a notion of inheritance based on static analysis.

8.2 An Example of Stack with Undo

We will illustrate the results of this chapter through the example in Figure 8.2. It defines a class `StackWithUndo` which extends the class `Stack` of Figure 7.1 by adding to the stack the capability of performing the *undo* of the last operation.

The comments in the figures are automatically derived when the framework presented in this chapter is instantiated with the Octagon abstract domain [86] refined with trace partitioning [59]. The class invariant for `StackWithUndo`, `SubInv`, states that the parent class invariant is still valid and moreover the field `undoType` cannot assume values outside the interval $[-1, 1]$. It implies that the method `undo` will never raise the exception

```
class StackWithUndo extends Stack {

  // SubInv : Inv, -1 <= undoType <= 1,
  //       if undoType == 1 then 0 < pos
  //   else if undoType == 0 then 0 <= pos <= size
  //   else if undoType == -1 then pos < size

  protected Object undoObject;
  protected int undoType;

  StackWithUndo(int x) {
    super(x);
    undoType = 0; undoObject = null;
  }

  void push(Object o) throws StackError {
    undoType = 1;
    super.push(o);
  }

  void pop() throws StackError {
    if(!isEmpty()) {
       undoType = -1;
       undoObject = stack[pos-1];
     }
    super.pop();
  }

     // StackError never thrown
  void undo() throws StackError {
    if(undoType == -1) {
      super.push(undoObject);
      undoType = 0;
    } else if (undoType == 1) {
      super.pop();
      undoType = 0;
    }
  }
}
```

Figure 8.2: `UndoStack`, an extension of `Stack` with undo

`StackErr`. Once again this information can be used for verification (if a class never raises an exception `Exc`, then the exceptional behavior described by `Exc` is never shown) and for optimization (as the exception handling can be dropped). Finally it is worth noting that `SubInv` has been obtained without accessing the parent code but just to its class invariant.

8.3 Non-Modular Analysis

Most object-oriented languages provide inheritance through class extension and differentiation: the programmer defines a subclass by writing down its parent class as well as the fields and the methods it adds. Moreover it is also allowed to redefine method bodies, and in the subclass one can gain access the previous implementation through the meta-variable `super`. For instance Java and C# have the syntactic construct `E extends B` which creates a subclass of `B`. With an abuse of language, the so-created subclass is called `E`, i.e. it takes the same name as the extension. However, in this thesis we differentiate between the name of the subclass and the name of the extension, so that `S` = `E extends B`, i.e. the proper subclass of `B` is called `S`.

In the rest of this section, we formally define the syntactic transformation behind `extends` and we show how to apply the results of previous sections in order to infer an invariant for the subclass.

8.3.1 Subclass Expansion

The Java subclassing mechanism can be formalized as a program transformation in the style of [13, 23, 90].

First, we define a class combination operator $\oplus \in [\text{Classes} \times \text{Classes} \to \text{Classes}]$ that forms a new class with fields, constructor and methods from its two arguments. If a method is defined in both classes then the value from the *left* argument is kept unchanged, whereas that from the *right* is kept after modifying its name by prefixing the right-argument class name. The new class constructor is that of the *left* argument. The *right* class constructor is renamed as in the case of overlapping methods and then kept.

Example 8.1 (Combination of Classes) Let us consider the class `Incr` $= \langle \texttt{init}, \{\texttt{a}\}, \{\texttt{add}\} \rangle$, where

$$\begin{aligned} \texttt{init} &= \lambda \texttt{a}_0.\ (\texttt{a} := \texttt{a}_0) \\ \texttt{add} &= \lambda().\ (\texttt{a} := \texttt{a} + 1). \end{aligned}$$

Now, let us extend the class Incr with: $\mathtt{E} = \langle \mathtt{init}, \{\mathtt{b}\}, \{\mathtt{add}\}\rangle$ where

$$\begin{aligned} \mathtt{init} &= \lambda(\mathtt{a}_0, \mathtt{c}_0).\ (\mathtt{super.init}(\mathtt{a}_0); \mathtt{b} := \mathtt{c}_0 - \mathtt{a}_0) \\ \mathtt{add} &= \lambda().\ (\mathtt{super.add}();\ \mathtt{b} := \mathtt{b} - 1). \end{aligned}$$

Then $\mathtt{E} \oplus \mathtt{Incr} = \langle \mathtt{init}, \{\mathtt{a}, \mathtt{b}\}, \{\mathtt{add},\ \mathtt{Incr\$add},\ \mathtt{Incr\$init}\}\rangle$.

The next step is to handle the binding of the meta-variable super. This is done by a renaming function $\theta \in [\text{Classes} \to \text{Classes}]$ that firstly replaces all the calls in the form of $\mathtt{super.m}(\ldots)$ with "superclassname"$\mathtt{\$m}(\ldots)$ and then substitutes all the occurrences of super with self.

Example 8.2 (Renaming Function, θ) When applying the renaming function to the previous example we obtain that $\theta(\mathtt{E} \oplus \mathtt{Incr}) = \langle \mathtt{init}, \{\mathtt{a}, \mathtt{b}\}, \{\mathtt{add},\ \mathtt{Incr\$add},\ \mathtt{Incr\$init}\}\rangle$, where

$$\begin{aligned} \mathtt{init} &= \lambda(\mathtt{a}_0, \mathtt{c}_0).\ (\mathtt{Incr\$init}(\mathtt{a}_0); \mathtt{b} := \mathtt{c}_0 - \mathtt{a}_0) \\ \mathtt{add} &= \lambda().\ (\mathtt{Incr\$add}(); \mathtt{b} := \mathtt{b} - 1). \end{aligned}$$

Finally, we assume that the name of methods in a class are implicitly prefixed by the class name, i.e. for a given class C the names $\mathtt{C\$m_i}$ and $\mathtt{m_i}$ are synonymous.

The program transformation that corresponds to the subclass extension mechanism is

$$\lambda\mathtt{E}.\ \lambda\mathtt{B}.\ \theta(\mathtt{E} \oplus \mathtt{B}),$$

i.e. the the subclass S corresponding to the Java/C# construct "E extends B" is $\mathtt{S} = \theta(\mathtt{E} \oplus \mathtt{B})$.

8.3.2 Analysis of the Expanded Class

Thanks to the syntactic transformation above, it is possible to directly apply the results of Chapter 5 for the inference of a subclass invariant. The idea is to take the code of the base class B, that of the extension E and then to instantiate (5.7) to the expanded class $\mathtt{S} = \theta(\mathtt{E} \oplus \mathtt{B})$.

The general form of S is $\langle \mathtt{init}, \mathtt{F_E} \cup \mathtt{F_B}, \{\mathtt{m_1} \ldots \mathtt{m_n}\} \cup \{\mathtt{n_1} \ldots \mathtt{n_k}\}\rangle$, where $\mathtt{F_B}$ are the fields of the base class, $\mathtt{m_i}$ are the methods of the base class *not* redefined by E and $\mathtt{F_E}$ and $\mathtt{n_i}$ are the fields and the methods from the extension E. As the redefined methods are hidden to the class clients, they are considered as they were protected methods. Thus they do not *contribute* to the class invariant. To sum up, a class invariant for S is a solution of the equation system (8.1) below. Note that for the sake of simplicity we omit the term $\overline{\text{Context}}$, i.e. we assume that the methods dos not expose the state

of the object. Nevertheless, the extension of our results to cope with this aspect are immediate, to the cost of a more complex notation.

$$Y = \bar{\mathbb{I}}[\![\mathtt{init_E}]\!] \bar{\sqcup} \bar{\bigsqcup}_{1 \leq i \leq n} \bar{\mathbb{M}}[\![\mathtt{m_i}]\!](Y) \bar{\sqcup} \bar{\bigsqcup}_{1 \leq i \leq k} \bar{\mathbb{M}}[\![\mathtt{n_i}]\!](Y). \tag{8.1}$$

However such a naive approach has several drawbacks. First, it is not suitable for the analysis of large class hierarchies. In fact, it is known [90, §6.4] that the expansion of the inheritance causes (in the worst case) a quadratic blow up of the code size. Therefore, the direct analysis of the expanded code may cause a quadratic loss of performances.

Second, in general `B` can be the base class for two distinct extensions `E` and `E′`. In that case the code `B` will be expanded and hence analyzed twice, with a further performances loss.

Third, in some cases the `B` source code is not available, e.g. with applications that use third-party libraries. In that case, the library providers are unlikely to distribute the source code. A reasonable solution may be to ship the class invariants together with the compiled code. In that way the analysis of `S` will use the source code of `E` and the class invariant for `B`, instead of its source.

8.4 Modular Analysis

In this section we present a modular approach to the inference of invariants for subclasses. The goal is to solve the equation (8.1) in a smarter way than performing the brute force fixpoint computation. In particular, we are interested in a solution that is a function of the base class invariant, so that the methods $\mathtt{m_i}$ does not need to be analyzed again. We consider separately the two orthogonal aspects of inheritance: class extension and method redefinition.

8.4.1 Class Extension

In this section we assume that `E` may add methods and fields to `B`, it may provide a new constructor, but it cannot redefine the methods of `B`.

The first remark is that the property $J_0 = \bar{\mathbb{I}}[\![\mathtt{init_E}]\!]$ is about the variables in $\mathtt{F_B} \cup \mathtt{F_E}$, so that it can be split in two parts $J_0 = J_0^{\mathtt{B}} \bar{\sqcup} J_0^{\mathtt{E}}$ where the first refers to the inherited fields and the latter to the fields of the extension $\mathtt{F_E}$. As for $J_0^{\mathtt{B}}$ is concerned, it is reasonable to assume $J_0^{\mathtt{B}} \bar{\sqsubseteq} I$, i.e. at creation time the subclass objects do not violate the superclass invariant. Recall that the order relation $\bar{\sqsubseteq}$ is the abstract counterpart of logical implication. This is a very common situation in object oriented programming: for example in

C++ [43] it is a standard procedure to call the superclass constructors to set up the inherited fields and then to initialize the fields in $\mathtt{F_E}$ and even Java semantics [56] forces the initialization of the base class fields before the subclass constructor(s) can access them.

Next, we look for a solution of (8.1) with a particular shape. Informally, S can behave either as the base class B or the extension E. Thus it is reasonable to look for a solution in the form of $I \bar{\square} U$, where I is the invariant of the base class B and U involves just the methods of E. Formally, U is a solution of the following recursive equation:

$$X = J_0^{\mathtt{E}} \bar{\square} \bar{\bigsqcup_{1 \leq i \leq k}} \bar{\mathbb{M}}[\![\mathtt{n_i}]\!](I \bar{\square} X). \tag{8.2}$$

As a consequence we obtain the following equation system:

$$\{(8.1), (8.2), J_0 = J_0^{\mathtt{B}} \bar{\square} J_0^{\mathtt{E}}, J = I \bar{\square} U\}. \tag{8.3}$$

It is worth noting that a solution J of (8.3) is a solution of (8.1), whereas in general the contrary does not hold.

The interest of using (8.3) is that the computation of J reduces to the computation of (8.2), so that the subclass invariant J can be obtained using just the superclass invariant (as $J = I \bar{\square} U$). Furthermore, it is possible to show that when analyzing a class hierarchy, in general the obtained speedup is linear in the number of direct descendants of a class and quadratic in the depth of the class hierarchy. The following theorem gives a sufficient and necessary condition for the existence of solutions of (8.1):

THEOREM 8.1 *If $\bar{\mathbb{M}}[\![\cdot]\!]$ is a join-morphism then the equation system (8.3) has a solution iff*

$$\bar{\bigsqcup_{1 \leq i \leq n}} \bar{\mathbb{M}}[\![\mathtt{m_i}]\!](U) \bar{\sqsubseteq} I \bar{\square} U. \tag{8.4}$$

Proof. Using the identities of (8.3) it is possible to rewrite the equation (8.1) as follows:

$$\begin{aligned}
J =& \bar{\mathbb{I}}[\![\mathtt{init_E}]\!] \bar{\square} \bar{\bigsqcup_{1 \leq i \leq n}} \bar{\mathbb{M}}[\![\mathtt{m_i}]\!](J) \bar{\square} \bar{\bigsqcup_{1 \leq i \leq k}} \bar{\mathbb{M}}[\![\mathtt{n_i}]\!](J) \\
=& J_0^{\mathtt{B}} \bar{\square} J_0^{\mathtt{E}} \bar{\square} \bar{\bigsqcup_{1 \leq i \leq n}} \bar{\mathbb{M}}[\![\mathtt{m_i}]\!](I \bar{\square} U) \bar{\square} \bar{\bigsqcup_{1 \leq i \leq k}} \bar{\mathbb{M}}[\![\mathtt{n_i}]\!](I \bar{\square} U) \\
=& J_0^{\mathtt{B}} \bar{\square} \bar{\bigsqcup_{1 \leq i \leq n}} \bar{\mathbb{M}}[\![\mathtt{m_i}]\!](I) \bar{\square} \bar{\bigsqcup_{1 \leq i \leq n}} \bar{\square} \bar{\mathbb{M}}[\![\mathtt{m_i}]\!](U) \bar{\square} J_0^{\mathtt{E}} \bar{\square} \bar{\bigsqcup_{1 \leq i \leq k}} \bar{\mathbb{M}}[\![\mathtt{n_i}]\!](I \bar{\square} U) \\
=& I \bar{\square} \bar{\bigsqcup_{1 \leq i \leq n}} \bar{\mathbb{M}}[\![\mathtt{m_i}]\!](U) \bar{\square} U
\end{aligned}$$

From basic lattice theory, it follows that the above equation is consistent with $J = I \square U$ iff (8.1) holds.

q.e.d.

If the subclass preserves the parent invariant w.r.t. the inherited fields, i.e. $\pi_{\mathtt{F_c}}(U) \bar{\sqsubseteq} I$ then the above equation system admits solutions. In general it may be difficult to prove the theorem hypothesis, and in the worst case it is computationally equivalent to solve (8.1). However, in Section 8.5 we will show that an analysis carried on using an instance of the $\mathcal{A}$-domain satisfies the hypothesis of the theorem, so that (8.1) must be checked once and for all.

Example 8.3 (Modular Inference of Subclass Invariant) The class `ClosedSystem` which models closed physical systems with a total energy $\mathtt{c}_0$ and two different kinds of internal energy `a` and `b`, can be defined as $\mathtt{ClosedSystem} = \langle \mathtt{init}, \{\mathtt{a}, \mathtt{b}\}, \{\mathtt{add}, \mathtt{sub}\}\rangle$, where

$$\begin{aligned} \mathtt{init} &= \lambda(\mathtt{a}_0, \mathtt{c}_0).\ (\mathtt{a} := \mathtt{a}_0;\ \mathtt{b} := \mathtt{c}_0 - \mathtt{a}_0) \\ \mathtt{add} &= \lambda().\ (\mathtt{a} := \mathtt{a} + 1;\ \mathtt{b} := \mathtt{b} - 1) \\ \mathtt{sub} &= \lambda().\ (\mathtt{a} := \mathtt{a} - 1;\ \mathtt{b} := \mathtt{b} + 1). \end{aligned}$$

Using (5.7) and the Octagon abstract domain it is possible to infer the class invariant $I = \{\mathtt{a} + \mathtt{b} = \mathtt{c}_0\}$. It states that the sum of the two fields is always equal to the value of the parameter passed to the constructor.

Now, let us consider the extension `ExtendedSystem` with a field `c`, a new constructor and a method `ext` :

$$\begin{aligned} \mathtt{init} &= \lambda(\mathtt{a}_0, \mathtt{c}_0).\ (\mathtt{super.init}(\mathtt{a}_0,\ \mathtt{c}_0);\ \mathtt{c} := \mathtt{c}_0) \\ \mathtt{ext} &= \lambda().\ (\mathtt{c} := \mathtt{c} - 1;\ \mathtt{a} := \mathtt{a} - 1). \end{aligned}$$

If we instantiate and solve (8.2) we obtain

$$\begin{aligned} U &= \{\mathtt{a} + \mathtt{b} = \mathtt{c},\ \mathtt{c} \leq \mathtt{c}_0\} \qquad \text{and} \\ J &= I \square U = \{\mathtt{a} + \mathtt{b} \leq \mathtt{c}_0,\ \mathtt{c} \leq \mathtt{c}_0\}. \end{aligned}$$

The invariant J is a class invariant for the class "`ExtendedSystem extends ClosedSystem`". Nevertheless, the direct application of (8.1) gives $J' = \{\mathtt{a} + \mathtt{b} = \mathtt{c}, \mathtt{c} \leq \mathtt{c}_0\}$. It is immediate to see that J' is a more precise invariant than J, that is $J' \bar{\sqsubseteq} J$.

The example above points out that whilst a solution of (8.3) is a solution of (8.1) in general it is not the *least* solution. Roughly, precision is lost because information flows from I to U but not in the other way.

8.4.2 Methods refining

In this section we consider a subclass that may "refine" the behavior of the superclass by redefining some of its methods. The modalities for doing it largely depend on the considered object oriented language. For example C++, C# and Java apply syntax criteria (the overriding method must have the same name and type as the overridden) whereas in Eiffel a method `n` overrides `m` if and only if the type of `n` is (co-variantly) a subtype of `m`. Nevertheless in order to be as language-independent as possible, we consider just how overriding and overridden methods interact and not how the overriding is provided by the language. However an implementation of an analyzer for a real language must consider this point.

As usual in abstract interpretation, we proceed by successive approximations. First we assume that all the methods of `S` may be executed, even those that are redefined, so that the results of the previous section apply directly. This is an over-approximation of the real behavior: if a method is redefined in a subclass, then it is not directly accessible by the context, but it is still reachable by the method bodies of `E`.

In general, because of late-binding, we must distinguish two situations: *down*calls and *up*calls: in the first case a method of the parent class invokes one that has been redefined and in the latter the interaction happens in the opposite direction. Once again, we can handle both by performing an upper-approximation. Let us consider a method definition in the form

$$\mathtt{m} = \lambda \mathtt{x_{in}}.\ (\ldots;\ \mathtt{v} := \mathtt{m_{call}}(\mathtt{y}); \ldots),$$

for some variables `v` and `y`.

If the invocation of $\mathtt{m_{call}}$ may resolve in a down-call, then a safe (but rather imprecise) approximation is to consider that an overriding method can arbitrarily modify the object internal state. Then the object state at the program point just after the assignment can be any $\sigma \in \Sigma$. Therefore when performing the analysis of the `m` body, the abstract environment just after the assignment will be set equal to $\alpha(\Sigma) = \bar{\top}$, the largest element of the abstract domain. An improvement would be to use I instead of $\alpha(\Sigma)$, but then the subclass must be checked to preserve the invariant I, i.e. $\pi_{\mathtt{F_B}}(J) \bar{\sqsubseteq} I$. In the next chapter we will consider a model of inheritance in which such a property holds.

On the other hand, if $\mathtt{m_{call}}$ resolves in an up-call then a worst case approximation of $\mathbb{M}[\![\mathtt{m_{call}}]\!]$ must be employed so that $\mathtt{v} := \bar{\mathbb{M}}[\![\mathtt{m_{call}}]\!](\bar{\top})$. Why this? Essentially because in the `m` body the (super)class invariant may not hold [81] so that it is not sound to assume it as an approximation of $\mathtt{m_{call}}$ semantics. Therefore, if we want to analyze the subclass code without referring to the

parent one then the only sound assumption is to consider the $\mathtt{m}_{\mathtt{call}}$ postcondition when its input is not known, i.e. it can be everything. This is sound as semantic functions are monotonic, so $\bar{\mathtt{d}} \bar{\sqsubseteq} \bar{\top} \Rightarrow \bar{\mathbb{M}}[\![\mathtt{m}_{\mathtt{call}}]\!](\bar{\mathtt{d}}) \bar{\sqsubseteq} \bar{\mathbb{M}}[\![\mathtt{m}_{\mathtt{call}}]\!](\bar{\top})$. Using the worst-case approximation, the base class must be shipped not only with the invariant I but also with an approximation $\bar{\mathbb{M}}[\![\mathtt{m}_{\mathtt{i}}]\!](\bar{\top})$ for each method $\mathtt{m}_{\mathtt{i}}$.

Example 8.4 (`StackWithUndo` Class Invariant) Applying the considerations above and instancing the equation system (8.3), it is possible to infer a class invariant for `StackWithUndo`. We use the Octagon abstract domain, so that $\bar{\sqcup}$ is the union of octagons. It is easy to see that the constructor preserves the `Stack` invariant, so

$$J_0 = I \bar{\sqcup} \{\mathtt{undoType} = 0\}.$$

Then we can consider the application of (8.2) to `undo` and to the overridden methods `push` and `pop`. The three perform an up-call. In that case it is sound to *replace* it with the `Stack` invariant as it holds at the program point just before the `super.pop` and `super.push` invocations. Therefore it is immediate to obtain

$$U = I \bar{\sqcup} \{-1 \leq \mathtt{undoType} \leq 1\},$$

so that using Th. 8.1 the class invariant for `StackWithUndo` is $J = I \bar{\sqcup} U$.

If it is needed, the obtained invariant and method post-conditions can be improved by using the incremental refinement technique of [50]. In our example, the abstract domain can be refined by partitioning it through the values assumed by `undoType`, that using the already computed class invariant J are at most 3. In such a way it is possible to infer the property:

$$\left\{ \begin{array}{l} \mathtt{undoType} = 1 \Rightarrow \mathtt{pos} > 0 \\ \mathtt{undoType} = 0 \Rightarrow 0 \leq \mathtt{pos} \leq \mathtt{size} \\ \mathtt{undoType} = -1 \Rightarrow \mathtt{pos} < \mathtt{size} \end{array} \right\}$$

so that it is proved that the method `undo` never throws the exception `StackError`.

8.5 Symbolic Relations and Inheritance

We now consider the case in which the underlying abstract domain is an instance of the $\mathcal{A}$-domain. The methods added or redefined by a subclass can be handled as in Chapter 7: their semantics is approximated by a suitable symbolic relation and the base class invariant is computed using the history-insensitive schema of Section 7.4.1. Then, if $I_{\mathtt{B}}$ is the history-insensitive class

invariant corresponding to the base class, then the subclass invariant in the form of $I_{\mathtt{B}} \curlyvee W$ can be given, with W defined as:

$$W = \pi_{\mathtt{F}'}(\mathtt{c}_{\mathtt{init_E}}) \curlyvee \bigvee_{1 \leq i \leq k} \pi_{\mathtt{F}'}(\mathtt{c}_{\mathtt{m_i}}).$$

It is routine to check that W satisfies the hypothesis of Theorem 8.1 so that $H = I_{\mathtt{B}} \curlyvee W$ is effectively a solution of (8.1) and hence a modular subclass invariant:

THEOREM 8.2 *Let* S *be a subclass of* B. *Furthermore, let* $I_{\mathtt{B}}$ *be an history-insensitive class invariant for* B. *Then* $H = I_{\mathtt{B}} \curlyvee W$ *is a class invariant for* S.

Last theorem is useful in that it guarantees that whenever we have an analysis that fulfills the requirements of being symbolic relational it can be immediately used to derive a subclass invariant without any additional check. For example this is the case of the Octagon domain used in the previous examples.

Furthermore, using symbolic relations the handling of up-calls can be improved: with reference to what we said in Section 8.4.2, we can use a symbolic relational approximation of the method $\mathtt{m}_{\mathtt{call}}$ instead of the worst-case one. Therefore, if the environment before the method invocation is approximated by $\mathtt{c}_F$, then the object state after that is approximated by $\pi_{F'}(\mathtt{c}_{\mathtt{m}_{\mathtt{call}}} \curlywedge \mathtt{c}_F)$. This because we have an approximation for the method input/output behavior w.r.t the object fields, $\mathtt{c}_{\mathtt{m}_{\mathtt{call}}}$, and one for the input, $\mathtt{c}_F$. Then we put together the two using $\curlyvee$, and we project on the output values $\mathtt{F}'$. In general this handling of up-calls is likely to be more precise than the one presented in Section 8.4.2.

Constraints may also be used to improve the treatment of down-calls: the overriding method(s) can be added as a further parameter to symbolic relations so that they assume the general form of $\mathtt{c}_{\mathtt{m}}[\mathtt{m}_{\mathtt{down}}]$. Therefore the resulting base class invariant $I[\mathtt{m}_{\mathtt{down}}]$ is parameterized by the (approximation of the) semantics of methods redefined in subclasses so that if $\mathtt{c}_{\mathtt{down}}$ is the constraint for the method overriding $\mathtt{m}_{\mathtt{down}}$ then H can be rewritten as

$$H = \pi_{F'}(K[\mathtt{m}_{\mathtt{down}}] \curlywedge \mathtt{c}_{\mathtt{down}}) \curlyvee W.$$

The formula above holds whenever the subclass S does not introduce new virtual functions, i.e. functions that can be overridden, and S subclasses do not further redefine $\mathtt{m}_{\mathtt{down}}$. Nevertheless, to the detriment of exposition clarity, it is possible to give a more general formulation to drop these two hypotheses.

8.6 Discussion

In this chapter we presented a framework for class modular analysis of object oriented languages in presence of inheritance. We instantiated the equations for the class invariants to cope with inheritance. We discussed the resolvability whenever a solution, function of the base class invariant, is required.

Chapter 9

Static Analysis-based Inheritance

Tu non pensavi ch'io loico fossi![1]

Dante Alighieri
Hell, Divine Comedy (1321)

In mainstream object oriented languages the inheritance relation is defined in terms of subtyping, i.e. a class `A` is a subclass of `B` if the type of `A` is a subtype of `B`. In this chapter we extend this notion to consider arbitrary class properties obtained by a modular static analysis of the class. In such a setting, the subclass relation boils down to the order relation on the abstract domain used for the analysis of the classes. We show how this approach yields a more semantic characterization of class hierarchies and how it can be used for an effective modular analysis of polymorphic code.

This chapter is based on the published work [75].

9.1 Behavioral Subtyping

Inheritance is one of the main features of object oriented languages. It allows a form of incremental programming and of code reuse. Moreover it allows the structuring of the code in a hierarchy, so that the classes composing a program (or a library) are organized in a subclass hierarchy. The traditional definition of the inheritance relation is that a class `A` is subclass of class `B` if its type is a subtype of that of `B`. Stated otherwise this means that an object that belongs to `A` can be used in any context that requires an object of `B`

[1] (Italian) You didn't think I was a logician!

without causing a type-error at runtime. However, the subtyping relation is not strong enough to ensure, for instance, that an object of `A` does not cause a division by zero, if the `B`'s object did not. Behavioral subtyping tries to overcome this problem [4, 73, 68].

Roughly speaking the behavioral subtype relationship guarantees that no unexpected behavior occurs when subtype objects replace supertype's ones. The essential idea is to annotate the class source code with a property in a suitable formal language. Such a property is called the behavior type of the class [73]. Then the behavioral subtyping relation is defined in terms of property implication: `A` is a subclass of `B` if its behavioral type implies that of `B`. The checking of this implication can be done in several ways: by a hand-proof [73], a theorem prover [68] or even at runtime [80]. However, most of the times the formal correspondence between the class semantics and the hand-provided behavioral type is neglected.

We present an approach to behavioral subtyping based on static analysis. The main idea is to analyze a class on a suitable abstract domain to infer a class invariant as well as methods preconditions and postconditions. We call the result of the analysis of `A` the observable of `A`, $\mathcal{O}(\texttt{A})$. An observable is a sound approximation of the class semantics, thus it is a behavioral type of `A`. The correspondence between the semantics of `A` and $\mathcal{O}(\texttt{A})$ is straightforwardly given by the soundness of the static analysis. The behavioral subtyping relation boils down to the order $\bar{\sqsubseteq}$ on the abstract domain: given two classes `A` and `B`, then $\mathcal{O}(\texttt{A}) \bar{\sqsubseteq} \mathcal{O}(\texttt{B})$ means that `A` preserves the behavior of `B`. In other words `A` is a behavioral subtype of `B`.

Our approach to behavioral subtyping has several advantages. First, as it is based on static analysis it does not require any human intervention for the annotation of the source code. Second, the observable is ensured to be a sound approximation of the class semantics and it saves programmer time. Third, as the order relation $\bar{\sqsubseteq}$ is decidable it can be automatically checked. Thus there is no need of using a theorem prover or to rely on unsound methods as runtime assertion monitoring [80]. Fourth, the definition of the behavioral subtyping in the abstract interpretation framework allows to use standard techniques as for instance domain refinement [31] in order to systematically improve the precision of the observables.

9.2 Examples

We illustrate our approach on the classes in Figure 9.1. They implement different kinds of bags. They have a method to add an element to the container, `add(e)` and `addSq(e)`, and to extract an element from it, `remove()`.

```
class Stack is
s : list of int;
init() : s = []
add(e) : s = e :: s
remove() : let s = e :: ls in
    s = ls;
    return e
```

(a) Stack

```
class Queue is
s : list of int;
init() : s = [];
add(e) : s = s :: e
remove() : let s = e :: ls in
    s = ls;
    return e
```

(b) Queue

```
class PosStack is
s : list of int;
init() : s = []
add(e) : s = |e| :: s
remove() : let s = e :: ls in
    s = ls;
    return e
```

(c) PosStack

```
class SqStack is
s : list of int;
init() : s = []
add(e) : s = |e| :: s
addSq(e) : s = (e ∗ e) :: s
remove() : let s = e :: ls in
    s = ls;
    return e
```

(d) SqStack

Figure 9.1: Four implementations of a bag

However, they differ in the handling of the elements: the method `remove()` of `Queue` returns the elements in the same order they have been inserted whereas that of `Stack`, `PosStack` and `SqStack` returns them in the reverse order. Moreover `PosStack` and `SqStack` contain only positive integers and `SqStack` has a further method that inserts the square of its argument. For the sake of simplicity we do not consider such errors, as removing an element from an empty container.

9.2.1 Class Hierarchy

It is evident that the four classes have different behaviors. However the three are not totally unrelated, so which is the relation between them? Which are the admissible class hierarchies? To put it another way, when is it safe to replace an object `s` of `Stack` with an object `q` of `Queue`? The answer depends on the meaning of "safe". In type theory "safe" means that the use of `q` at the place of `s` will not cause a run-time type error, if `s` did not cause one [90]. Thus in the example, the classes `Stack`, `PosStack` and `Queue` have the same type so that for instance `Stack` may be a subtype of `Queue` and conversely. Only `SqStack` has to be a subtype of `Stack`, `PosStack` or `Queue`, due to method `addSq`. This is the only constraint on the possible class hierarchies.

On the other hand, if the context requires that the values extracted from the bag are in the reverse order with respect to the insertion one then it is not "safe" anymore to replace `s` with `q`. Thus the order of the elements of a bag is a property, different from types, that induces a different inheritance relationship. In particular, from this point of view `Stack` and `SqStack` exhibit the same property, so that `Stack` may be defined as a subclass of `SqStack`. Therefore, the admissible class hierarchies are different from that allowed by the subtyping relation.

9.2.2 Systematic Refinement of the Class Hierarchy

Types and element ordering are both properties of classes that can be discovered once they are analyzed on suitable abstract domains, say $\bar{\mathsf{T}}$ and $\bar{\mathsf{S}}$ respectively. The two domains can be combined together using the reduced product $\bar{\mathsf{P}} = \bar{\mathsf{T}} \otimes \bar{\mathsf{S}}$ (cf. Definition 2.4). Thus, using the more precise abstract domain $\bar{\mathsf{P}}$ it is possible to infer more precise class properties. In the example `SqStack` can only be a subclass of `Stack` or of `PosStack`. However it is still admissible to have `Stack` subclass of `PosStack` and *vice versa*. This is essentially a consequence of the fact that $\bar{\mathsf{P}}$ does not capture the sign of the elements in field `s`. Therefore, $\bar{\mathsf{P}}$ can be combined with the Sign abstract domain [29] in order to capture such a property: $\bar{\mathsf{R}} = \bar{\mathsf{P}} \otimes \mathsf{Sign}$. Thereafter, using $\bar{\mathsf{R}}$ we obtain that `Stack` can never be subclass of `PosStack` as it does not preserve the property that all the elements in `s` are positive integers. Only four class hierarchies preserve the properties captured by $\bar{\mathsf{R}}$: the trivial one in which the subclass relation is the identity and the three listed in Figure 9.2.

9.2.3 Modular Verification

The initial motivation of our work on behavioral suptyping was the application of behavioral subtyping to the modular analysis of polymorphic object oriented code, robust with respect to the addition of subclasses. Consider for example the following function that references an object of type `PosStack`:

$$\texttt{sqrt(PosStack p)} \ : \ \texttt{return} \ \sqrt{\texttt{p.remove()}}.$$

One would like to prove `sqrt` correct for all possible future subclasses of `PosStack`, as all these may be passed as a parameter. This is possible if the subclasses do not violate the `PosStack` property that the elements are always positive. It is evident that the subtyping-based subclass relation is too weak to ensure this property.

However, if only subclass relations based on the properties encoded in

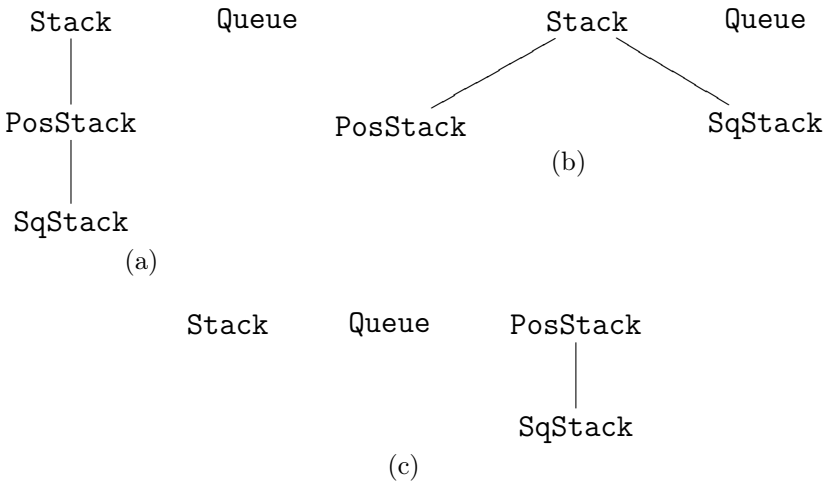


Figure 9.2: Admissible class hierarchies using $\bar{\mathsf{R}}$

$\bar{\mathsf{R}}$ are allowed, then all the subclasses of `PosStack` preserve the required invariants. This reduces the proof that `sqrt` never performs the square root of a negative number to proving it with `PosStack` as an argument.

9.3 Observables

A behavioral type of a class `A` is a property of the semantics of `A` [73]. Next, we consider behavioral types that are the result of a static analysis of `A`, and we define them as *observables* of the class: $\mathcal{O}(\mathtt{A}) = \bar{\mathbb{C}}[\![\mathtt{A}]\!]$.

The advantage of defining the behavioral type of a class as the result of a static analysis of its source is that it can be automatically inferred. Moreover, given two classes `A` and `B`, it is sufficient to check if $\mathcal{O}(\mathtt{A}) \bar{\sqsubseteq} \mathcal{O}(\mathtt{B})$ in order to verify if they are in the behavioral subtype relation. In a static analysis the elements of $\bar{\mathsf{D}}$ are computer-representable approximations of the concrete properties and the $\bar{\sqsubseteq}$ order on the abstract domain $\bar{\mathsf{D}}$ is a *decidable* abstract counterpart for the logical implication. Therefore the behavioral subtype relation is decidable too.

9.3.1 Domain of Observables

We begin by defining the abstract domain of observables. Let us put ourselves in the setting of Theorem 5.1. We recall that a solution of (5.7) is a tuple $\langle \bar{\mathsf{I}}, \bar{\mathsf{I}}_0, \bar{\mathsf{I}}_1 \ldots \bar{\mathsf{I}}_n \rangle \in \bar{\mathsf{D}}^{n+2}$ where the first component is a class invariant, $\bar{\mathsf{I}}_0$ is the constructor postcondition and the $\bar{\mathsf{I}}_i$s, $i \geq 1$ are the methods postconditions. The method preconditions can be obtained using a backward analysis starting

from the postcondition: $\bar{\mathsf{P}}_i = \bar{\mathbb{M}}^{<}[\![\mathtt{m_i}]\!](\bar{\mathsf{I}}_i)$. Therefore, the result of a static analysis of `A` is:

$$\bar{\mathbb{C}}[\![\mathtt{A}]\!] = \langle \bar{\mathsf{I}}, \{\mathtt{m_i} : \bar{\mathsf{P}}_i \to \bar{\mathsf{I}}_i \mid \mathtt{m}_i \in \{\mathtt{init}\} \cup \mathtt{M}\}\rangle.$$

The domain of the observables, $\langle \bar{\mathsf{O}}, \bar{\sqsubseteq}_o\rangle$, is built on the top of the domain used for the analysis, i.e. $\langle \bar{\mathsf{D}}, \bar{\sqsubseteq}\rangle$. The elements belong to the set

$$\bar{\mathsf{O}} = \{\langle \bar{\mathsf{I}}, \{\mathtt{m}_i : \bar{\mathsf{P}}_i \to \bar{\mathsf{I}}_i\}\rangle \mid \bar{\mathsf{I}} \in \bar{\mathsf{D}}, \forall i.\ \bar{\mathsf{P}}_i, \bar{\mathsf{I}}_i \in \bar{\mathsf{D}}\}.$$

We tacitly assume that if a method `n` is not defined in a class, then its precondition and postconditions are respectively $\bar{\top}$and $\bar{\bot}$. Such an assumption allows us to give a smart definition of the order relation $\bar{\sqsubseteq}_o$:

Definition 9.1 (Order on Observables, $\bar{\sqsubseteq}_o$) *Let* $\mathsf{o_1} = \langle \bar{\mathsf{I}}, \{\mathtt{m_i} : \bar{\mathsf{P}}_i \to \bar{\mathsf{I}}_i\}\rangle$ *and* $\mathsf{o_2} = \langle \bar{\mathsf{J}}, \{\mathtt{m_j} : \bar{\mathsf{Q}}_j \to \bar{\mathsf{J}}_j\}\rangle$ *be two elements*[1] *of* $\bar{\mathsf{O}}$. *Then*

$$\mathsf{o_1} \bar{\sqsubseteq}_o \mathsf{o_2} \iff \bar{\mathsf{I}} \bar{\sqsubseteq} \bar{\mathsf{J}} \wedge (\forall \mathtt{m}_i.\ \bar{\mathsf{Q}}_i \bar{\sqsubseteq} \bar{\mathsf{P}}_i \wedge \bar{\mathsf{I}}_i \bar{\sqsubseteq} \bar{\mathsf{J}}_i).$$

Roughly speaking, if $\mathsf{o_1}$ and $\mathsf{o_2}$ are the observables of two classes `A` and `B` then the order $\bar{\sqsubseteq}_o$ ensures that `A` preserves the class invariant of `B` and that the methods of `A` are a "safe" replacement of those with the same name in `B`. Intuitively, the precondition condition says that if the context satisfies Q_i then it satisfies the inherited method precondition P_i too. Thus the inherited method can be used in any context where its ancestor can. On the other hand, the postcondition of the inherited method may be stronger than that of the ancestor.

Having defined $\bar{\sqsubseteq}_o$, it is routine to check that if $\bar{\mathsf{D}}$ is a complete lattice then $\bar{\bot}_o = \langle \bar{\bot}, \{\mathtt{m_i} : \bar{\top} \to \bar{\bot}\}\rangle$ is the smallest element of $\bar{\mathsf{O}}$ and $\bar{\top}_o = \langle \bar{\top}, \{\mathtt{m_i} : \bar{\bot} \to \bar{\top}\}\rangle$ is the largest one.

The join and the meet operations can be defined point-wise:

Definition 9.2 (Join and Meet of Observables, $\bar{\sqcup}_o$ and $\bar{\sqcap}_o$) *Let* $\mathsf{o_1} = \langle \bar{\mathsf{I}}, \{\mathtt{m_i} : \bar{\mathsf{P}}_i \to \bar{\mathsf{I}}_i\}\rangle$ *and* $\mathsf{o_2} = \langle \bar{\mathsf{J}}, \{\mathtt{m_j} : \bar{\mathsf{Q}}_j \to \bar{\mathsf{J}}_j\}\rangle$ *be two elements of* $\bar{\mathsf{O}}$. *Then*

$$\mathsf{o_1} \bar{\sqcup}_o \mathsf{o_2} = \langle \bar{\mathsf{I}} \bar{\sqcup} \bar{\mathsf{J}}, \{\mathtt{m}_i : \bar{\mathsf{Q}}_i \bar{\sqcap} \bar{\mathsf{P}}_i \to \bar{\mathsf{I}}_i \bar{\sqcup} \bar{\mathsf{J}}_i\}\rangle$$
$$\mathsf{o_1} \bar{\sqcap}_o \mathsf{o_2} = \langle \bar{\mathsf{I}} \bar{\sqcap} \bar{\mathsf{J}}, \{\mathtt{m}_i : \bar{\mathsf{Q}}_i \bar{\sqcup} \bar{\mathsf{P}}_i \to \bar{\mathsf{I}}_i \bar{\sqcap} \bar{\mathsf{J}}_i\}\rangle$$

Moreover, let us suppose that the order relation $\bar{\sqsubseteq}$ is decidable. For instance, this is the case of an abstract domain used for an effective static analysis. As $\bar{\sqsubseteq}_o$ is defined in terms of $\bar{\sqsubseteq}$ and the universal quantification ranges on a finite number of methods then $\bar{\sqsubseteq}_o$ is decidable too. To sum up, we have the following result:

[1] We use the same index for methods with the same name. For instance P_i and Q_i are the preconditions for the homonym method $\mathtt{m_i}$ of $\mathsf{o_1}$ and $\mathsf{o_2}$.

THEOREM 9.1 *Let* $\langle \bar{\mathsf{P}}, \bar{\sqsubseteq}, \bar{\bot}, \bar{\top}, \bar{\sqcup}, \bar{\sqcap} \rangle$ *be a complete lattice. Then* $\langle \bar{\mathsf{O}}, \bar{\sqsubseteq}_o, \bar{\bot}_o, \bar{\top}_o, \bar{\sqcup}_o, \bar{\sqcap}_o \rangle$ *is a complete lattice. Moreover, if* $\bar{\sqsubseteq}$ *is decidable then* $\bar{\sqsubseteq}_o$ *is decidable too.*

Proof. It is straightforward to check that, by definition, $\bar{\bot}_o$and $\bar{\top}_o$ are respectively the least and the largest element of $\bar{\mathsf{O}}$ w.r.t. the order $\bar{\sqsubseteq}_o$. For the join, consider a set $\{\mathtt{o}_k\}$ of observables. Then by definition of $\bar{\sqcup}_o$,

$$\bar{\bigsqcup}_o \{\mathtt{o}_i\} = \left\langle \bar{\bigsqcup}_k \bar{\mathsf{I}}_k, \left\{ \mathtt{m}_i : \bar{\prod}_k \bar{\mathsf{Q}}_{i_k} \to \bar{\bigsqcup}_k \bar{\mathsf{I}}_{i_k} \right\} \right\rangle$$

exists as, by hypothesis, $\bar{\sqcup}$ and $\bar{\sqcap}$ are respectively a complete join-morphism and a complete meet-morphism. For the same reason $\bar{\sqcap}_o$ is a complete meet-morphism.

q.e.d.

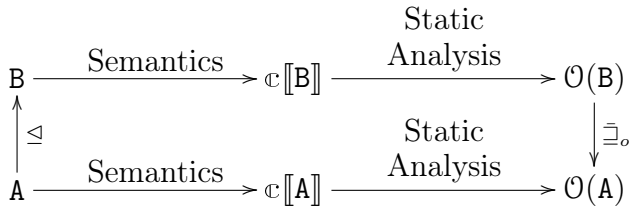
9.4 Subclassing through Observables

It is now possible to formally give the definition of subclassing as inclusion relation between elements of the abstract domain:

DEFINITION 9.3 *Let* $\mathtt{A}$ *and* $\mathtt{B}$ *be two classes,* $\langle \bar{\mathsf{O}}, \bar{\sqsubseteq} \rangle$ *a domain of observables. Then* $\mathtt{A}$ *is a subclass of* $\mathtt{B}$*,* $\mathtt{A} \trianglelefteq \mathtt{B}$*, with respect to the properties encoded by* $\bar{\mathsf{O}}$ *iff* $\mathcal{O}(\mathtt{A}) \; \bar{\sqsubseteq}_o \; \mathcal{O}(\mathtt{B})$.

Observe that when $\langle \bar{\mathsf{O}}, \bar{\sqsubseteq}_o \rangle$ is instantiated with the types abstract domain [26] then the relation defined above coincides with the traditional subtyping-based definition of subclassing [16].

The Definition 9.3 can be visualized by the following diagram:



This diagram essentially shows how the concept of subclassing is linked to the semantics of classes. It states that when the abstract semantics of $\mathtt{A}$ and $\mathtt{B}$ are compared, that of $\mathtt{A}$ implies the one of $\mathtt{B}$. That means that $\mathtt{A}$ refines $\mathtt{B}$ w.r.t. the properties encoded by the abstract domain $\bar{\mathsf{O}}$. This is in accord

with the mundane understanding of inheritance which states that a subclass is a specialization of the ancestor [81].

Moreover we have made no hypothesis on the abstraction of the concrete semantics. In particular we do not differentiate between history properties and state properties, unlike [73], the two being just different abstractions of the concrete semantics. In fact, history properties correspond to trace abstractions and state properties to state abstractions.

It is worth noting that our definition of observables is slightly different from the notion of observables of [21]. In that paper, the authors present a theory of observables for logic programming. In particular they define an observable as an abstraction *function*. Our approach is different in that we define observables as abstract *elements*.

9.4.1 Static Checking of Behavioral Subtyping

The main advantage of our approach is that the subclassing relation can automatically be checked by a compiler: the derivation of class observables is automatic and their inclusion is decidable. As a consequence a compiler can accept subclasses only if they preserve the parent *behavior*. For instance, this is in the spirit of Eiffel subclassing mechanism [80]. However, the specification of Eiffel requires to check the preservation of the ancestor invariants at runtime. An interesting future work can be the extension of our work on subclassing to the Eiffel language.

9.4.2 Modular Verification

A major advantage of having the compiler which rejects subclass definitions, that do not preserve the parent properties, is that it enables a form of modular analysis for polymorphic functions. Consider the following polymorphic function `f`, that references an object of type `B`:

$$\texttt{f(B b)} \ : \ \ldots \ \texttt{b.m(v)} \ldots.$$

Now, suppose to analyze it on the $\langle \bar{\mathsf{D}}, \bar{\sqsubseteq} \rangle$ abstract domain. If the analysis is performed using `B` then the call `b.m(v)` resolves to the invocation of the method $\mathtt{m}_{\mathtt{B}}$ of `B`. Having an observable of `B`, the precondition $\bar{\mathsf{Q}}$ and the post-condition $\bar{\mathsf{J}}$ of $\mathtt{m}_{\mathtt{B}}$ can be used in the analysis, so that the body of the methods does not need to be analyzed again. Thus, if $\bar{\mathsf{v}} \in \bar{\mathsf{D}}$ is the approximation of the concrete values taken by the variable `v` then

$$\bar{\mathsf{v}} \bar{\sqsubseteq} \bar{\mathsf{Q}} \Longrightarrow \bar{\mathbb{M}}[\![\mathtt{m}_{\mathtt{B}}]\!](\bar{\mathsf{v}}) \bar{\sqsubseteq} \bar{\mathsf{J}}.$$

The result of such an analysis is valid for all the invocations `f(a)` where `a` is an instance of a class $\mathtt{A} \trianglelefteq \mathtt{B}$. This can be shown as follows. If $\mathtt{A} \trianglelefteq \mathtt{B}$ then

$\mathcal{O}(\mathtt{A}) \bar{\sqsubseteq}_o \mathcal{O}(\mathtt{B})$. Then, by definition of the order relation $\bar{\sqsubseteq}_o$ the method $\mathtt{m_A}$ of $\mathtt{A}$ is such that $\mathtt{m_A} : \bar{\mathsf{P}} \to \bar{\mathsf{I}}$ with $\bar{\mathsf{Q}} \bar{\sqsubseteq} \bar{\mathsf{P}}$ and $\bar{\mathsf{I}} \bar{\sqsubseteq} \bar{\mathsf{J}}$. So:

$$\bar{\mathsf{v}} \bar{\sqsubseteq} \bar{\mathsf{Q}} \bar{\sqsubseteq} \bar{\mathsf{P}} \Longrightarrow \bar{\mathbb{M}}[\![\mathtt{m_A}]\!](\bar{\mathsf{v}}) \bar{\sqsubseteq} \bar{\mathsf{I}} \wedge \bar{\mathsf{I}} \bar{\sqsubseteq} \bar{\mathsf{J}}.$$

Thus $\bar{\mathsf{J}}$ is a sound approximation of the semantics of the method $\mathtt{m_A}$. As a matter of fact we have proved the following theorem:

THEOREM 9.2 *Let* $\mathtt{A}$ *and* $\mathtt{B}$ *two classes such that* $\mathtt{A} \trianglelefteq \mathtt{B}$. *Let* $\mathtt{m_B}$ *be a method of* $\mathtt{B}$ *and* $\mathtt{m_A}$ *the homonym method that belongs to* $\mathtt{A}$. *Let* $\mathtt{m_B} : \bar{\mathsf{Q}} \to \bar{\mathsf{J}}$ *and* $\mathtt{m_A} : \bar{\mathsf{P}} \to \bar{\mathsf{I}}$. *Then*

$$\forall \bar{\mathsf{v}} \bar{\sqsubseteq} \bar{\mathsf{P}}.\ \bar{\mathsf{v}} \bar{\sqsubseteq} \bar{\mathsf{Q}} \Longrightarrow \bar{\mathbb{M}}[\![\mathtt{m_A}]\!](\bar{\mathsf{v}}) \bar{\sqsubseteq} \bar{\mathsf{J}}.$$

Hence the analysis of polymorphic code using the superclass is enough to state that the result is valid for all the subclasses. So, it is not necessary to reanalyze the code for each subclass of $\mathtt{B}$.

9.4.3 Domain Refinement

A further advantage of formalizing the behavioral subtyping in the abstraction interpretation framework is that it is possible to apply well-known abstract domain refinement techniques [31, 54, 52] in order to improve the precision of the observables. Hence having more fine-grain class hierarchies. In particular, the use of the reduced product is practical for refining the precision of the captured properties.

THEOREM 9.3 (DOMAIN REFINEMENT) *Let* $\langle \bar{\mathsf{D}}_1, \bar{\sqsubseteq}_1 \rangle$ *and* $\langle \bar{\mathsf{D}}_2, \bar{\sqsubseteq}_2 \rangle$ *be two abstract domains such that*

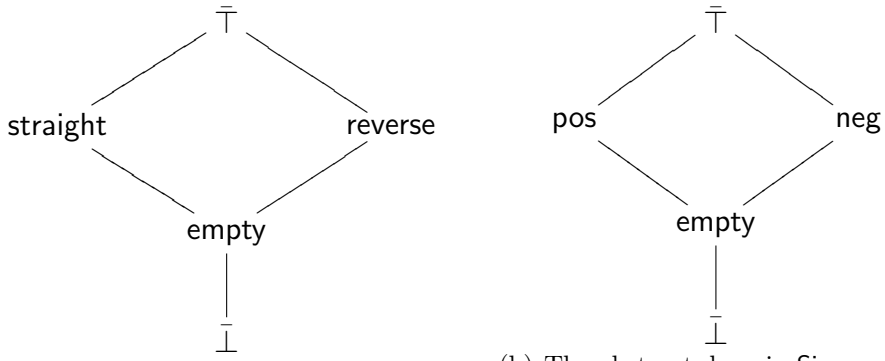
$$\langle \bar{\mathsf{D}}_1, \bar{\sqsubseteq}_1 \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{\mathsf{D}}_2, \bar{\sqsubseteq}_2 \rangle.$$

If $\mathtt{A} \trianglelefteq \mathtt{B}$ *using* $\bar{\mathsf{D}}_1$, *then* $\mathtt{A} \trianglelefteq \mathtt{B}$ *using* $\bar{\mathsf{D}}_2$.

Proof. By theorem hypotheses $\mathcal{O}(\mathtt{A}) \bar{\sqsubseteq}_o \mathcal{O}(\mathtt{B})$, where $\bar{\sqsubseteq}_o$ is the order built on the top of $\bar{\sqsubseteq}_1$. By monotonicity of Galois connections, $\alpha(\mathcal{O}(\mathtt{A})) \bar{\sqsubseteq}'_o \alpha(\mathcal{O}(\mathtt{B}))$, where $\bar{\sqsubseteq}'_o$ is the order built on the top of $\bar{\sqsubseteq}_2$. If $\bar{\mathbb{C}}[\![\cdot]\!]$ is the best abstract function defined on $\bar{\mathsf{D}}_2$ [29], then by soundness of static analyses $\alpha(\mathcal{O}(\mathtt{A})) \bar{\sqsubseteq}'_o \bar{\mathbb{C}}[\![\mathtt{A}]\!]$ and $\alpha(\mathcal{O}(\mathtt{B})) \bar{\sqsubseteq}'_o \bar{\mathbb{C}}[\![\mathtt{B}]\!]$ imply that $\mathtt{A} \trianglelefteq \mathtt{B}$ using $\bar{\mathsf{D}}_2$.

q.e.d.

An abstract domain of observables must, at least, encapsulate the types abstract domain $\bar{\mathsf{T}}$. On the other hand we have argued before how a further



(a) The abstract domain $\bar{\mathsf{S}}$ (b) The abstract domain Sign

Figure 9.3: Abstracts domains expressing the order of elements and their sign

abstract domain $\bar{\mathsf{D}}$ is needed to capture non-typing properties, e.g. the sign of the field values. Then the domain of observables can be built on the top of the reduced product of the two: $\bar{\mathsf{P}} = \bar{\mathsf{T}} \otimes \bar{\mathsf{D}}$. As a consequence, from a well-known result in abstract interpretation (cf. Theorem 10.1.0.2 of [31]) it follows that $\bar{\mathsf{P}}$ is a domain more precise than types, so that in general the resulting $\trianglelefteq$ relation is more precise than the subtyping one.

9.5 Application to the Examples

In this section we show how the definition of the previous section applies to the examples of Sect. 9.2. At first we show how when instantiating the underline abstract domain with types, the definition of $\trianglelefteq$ reduces to the traditional subtype-based one.

It is known that types can be seen as an abstract interpretation [26]. We consider the abstract domain $\bar{\mathsf{T}}$ corresponding to the Church/Curry monotypes. Then, if we instantiate the Definition 9.3 with the abstract domain $\bar{\mathsf{T}}$ we obtain that:

$$\begin{aligned}\mathcal{O}(\texttt{Stack}) = \mathcal{O}(\texttt{Queue}) = \mathcal{O}(\texttt{PosStack}) = \\ \{\langle \texttt{s} : \texttt{list of int}\rangle, \\ \{\texttt{init} : \texttt{void} \to \texttt{void};\ \texttt{add} : \texttt{int} \to \texttt{void}; \\ \texttt{remove} : \texttt{void} \to \texttt{int}\}\},\\ \mathcal{O}(\texttt{SqStack}) = \{\langle \texttt{s} : \texttt{list of int}\rangle,\ \{\texttt{init} : \texttt{void} \to \texttt{void}; \\ \texttt{add},\ \texttt{addSq} : \texttt{int} \to \texttt{void};\ \texttt{remove} : \texttt{void} \to \texttt{int}\}\}\end{aligned}$$

Therefore the only constraint on the definition of the subclassing relation

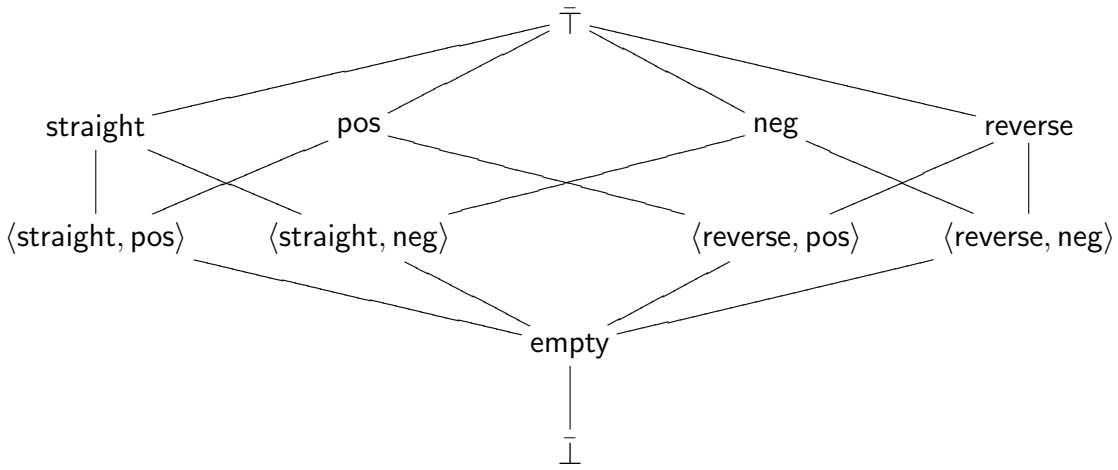


Figure 9.4: The abstract domain $\bar{\mathsf{H}} = \bar{\mathsf{S}} \otimes \mathsf{Sign}$

is that `SqStack` cannot be the ancestor of any of the other three. This is because $\mathcal{O}(\texttt{SqStack})$ is a subtype of $\mathcal{O}(\texttt{Stack})$ [16].

A different subclassing relation can be obtained using the abstract domain in Figure 9.3(a), whom intuitive meaning is to consider if the elements of the list are inserted at the head or tail position. It is worth noting that the order of the elements is a history property. In that case, the observables using $\bar{\mathsf{S}}$ are:

$$\begin{aligned}
\mathcal{O}(\texttt{Stack}) = \mathcal{O}(\texttt{PosStack}) = \{&\langle \texttt{s} : \mathsf{reverse}\rangle,\\
&\{\texttt{init} : \bar{\bot} \to \mathsf{empty};\ \texttt{add} : \mathsf{reverse} \to \mathsf{reverse};\\
&\quad \texttt{remove} : \mathsf{reverse} \to \mathsf{reverse}\}\},\\
\mathcal{O}(\texttt{SqStack}) = \{&\langle \texttt{s} : \mathsf{reverse}\rangle, \{\texttt{init} : \bar{\bot} \to \mathsf{empty};\\
&\quad \texttt{add},\ \texttt{addSq} : \mathsf{reverse} \to \mathsf{reverse};\\
&\quad \texttt{remove} : \mathsf{reverse} \to \mathsf{reverse}\}\},\\
\mathcal{O}(\texttt{Queue}) = \{&\langle \texttt{s} : \mathsf{straight}\rangle, \{\texttt{init} : \bar{\bot} \to \mathsf{empty};\\
&\quad \texttt{add} : \mathsf{straight} \to \mathsf{straight};\\
&\quad \texttt{remove} : \mathsf{straight} \to \mathsf{straight}\}\}.
\end{aligned}$$

In this last example, it happens that for instance `Queue` and `Stack` can never be in the subclass relation as neither $\mathcal{O}(\texttt{Queue}) \bar{\sqsubseteq}_o \mathcal{O}(\texttt{Stack})$ nor $\mathcal{O}(\texttt{Stack}) \bar{\sqsubseteq}_o \mathcal{O}(\texttt{Queue})$.

However nothing avoids to have $\texttt{Stack} \trianglelefteq \texttt{PosStack}$ as $\bar{\mathsf{S}}$ do not capture the sign of the elements in `s`, but just the order in which they are inserted. Therefore it is possible to refine $\bar{\mathsf{S}}$ using the domain Sign of Figure 9.3(b). In that case we consider the domain $\bar{\mathsf{H}}$ of Figure 9.4 that is the reduced product

of the two: $\bar{\mathsf{H}} = \bar{\mathsf{S}} \otimes \mathsf{Sign}$. Thus the resulting observables for the two classes are:

$$\begin{aligned}
\mathcal{O}(\texttt{Stack}) = \{&\mathtt{s} : \mathsf{reverse}, \{\mathtt{init} : \bar{\perp} \rightarrow \mathsf{empty};\\
&\mathtt{add} : \mathsf{reverse} \rightarrow \mathsf{reverse};\\
&\mathtt{remove} : \mathsf{reverse} \rightarrow \mathsf{reverse}\}\},\\
\mathcal{O}(\texttt{PosStack}) = \{&\mathtt{s} : \langle \mathsf{reverse},\ \mathsf{pos}\rangle, \{\mathtt{init} : \bar{\perp} \rightarrow \mathsf{empty};\\
&\mathtt{add} : \langle \mathsf{reverse},\ \mathsf{pos}\rangle \rightarrow \langle \mathsf{reverse},\ \mathsf{pos}\rangle;\\
&\mathtt{remove} : \langle \mathsf{reverse},\ \mathsf{pos}\rangle \rightarrow \langle \mathsf{reverse},\ \mathsf{pos}\rangle\}\}.
\end{aligned}$$

It is routine to check that $\texttt{Stack} \not\trianglelefteq \texttt{PosStack}$.

Eventually, the subclass relation that brings to the class hierarchies in Figure 9.2 is obtained considering the properties encoded by the abstract domain $\bar{\mathsf{R}} = \bar{\mathsf{T}} \otimes \bar{\mathsf{H}} = \bar{\mathsf{T}} \otimes \bar{\mathsf{S}} \otimes \mathsf{Sign}$.

9.6 Discussion

In this chapter we have presented an approach to the behavioral subtyping based on a modular static analysis of classes' source. In particular we have shown how the subclassing relation can be defined in terms of the order on the underlying abstract domain. Our approach has several advantages over traditional subtyping and behavioral subtyping: the class behavioral type is automatically inferred, the subtyping is decidable, it is more semantically characterized and it is formulated in the abstract interpretation framework so that well-known techniques on the composition and refinement of abstracts domain can be used. Moreover, in this setting the problem of analyzing, and hence verifying, polymorphic code is by far more simple. In fact as every subclass extension preserves, by definition, the ancestor invariant then it is not required to reanalyze the code once a new subclass is defined.

However, an open problem is the choice of the abstract domain used for the inference of observables. For what concerns general purposes object oriented languages, it is difficult to fix in advance the properties that one wishes to be preserved by subclasses. Types have been shown to be effective for that purpose, so that an abstract domain of observables must at least include them. One can think to continue in that direction considering other runtime errors, e.g. division by zero, overflow or null-pointer dereferencing, so that given a class, all its subclasses are assured not to introduce runtime errors.

On the other hand we are trustful that our approach can be effective for the design and the development of problem-specific object oriented languages. As an example, let us consider a language for smartcards programming. In

this setting security is important, so that a wished property is that if a subclass does not reveal a secret, so do the subclasses. In that case, we can use a domain of observables able to capture security and information-flow properties. Embedded systems are another field that may take advantage of a more constrained subclass relation. In fact, in such a field it is immediate to see the benefits of having a language that ensures that subclasses does not violate the space and time constraints of the superclass.

In the future we plan to extend this work to cope with multiple inheritance and Java interfaces, too. The first extension is quite straightforward. The case of interfaces is more difficult: an interface is essentially a type specification, though most of the time such a specification is not expressive enough. Consider for example the case of a Java thread, which can be defined using either the `Runnable` interface or the `Thread` class. In both cases the class implementing a thread needs to define a method `run`. So what is the difference between a class implementing `Runnable` or extending `Thread`? The intuition is that in both cases the behavioral type of the class is the same, the difference being just syntactic. For instance, in [14] it is clearly hinted that the use of the `Runnable` is just a way to overcome the problem of single inheritance. We plan to define a specification language in order to cope with not-typing properties able to express properties imposed by interfaces. Then we will use it to prove that a class correctly implements an interface.

Chapter 10

Context Approximation with Regular Expressions

Sì che raffigurar m'è più latino.[1]

Dante Alighieri
Paradise, Divine Comedy (1321)

In this chapter we face the problem of analyzing mutual recursive classes by presenting a separate compositional analysis for object-oriented languages. In particular, we show how a generic static analysis of a context that uses an object can be split into two separate semantic functions involving respectively only the context and the object. The fundamental idea is to use a regular expressions for approximating the interactions between the context and the object. Then, we introduce an iterative schema for composing the two semantic functions. The iteration process returns at each step an upper approximation of the concrete semantics, so that the iterations can be stopped as soon as the desired degree of precision is reached. Furthermore, the analysis can be easily parallelized enabling a gain in time and memory. Finally, we illustrate our approach with a core object-oriented language with aliasing.

This chapter is based on the published work [78].

10.1 Introduction

One important facet of object-oriented design is encapsulation [81]. Encapsulation hides the objects' inner details from the outside world and allows a

[1] (Italian) So that my understanding become easier.

hierarchical structuring of code. As a consequence, a program written in the object-oriented style has often the structure of $\mathtt{C}[\mathtt{o}]$, where $\mathtt{C}[\cdot]$ is a context which interacts with an encapsulated object $\mathtt{o}$.

In this chapter we are interested in exploiting the encapsulation features of the object-oriented languages for obtaining an efficient static analysis, namely to separately analyze the context and the object. Most available analyses are not separated e.g. [92, 93], or they are imprecise as they assume the worst case for the calling context, e.g. [8, 96]. A separate analysis presents several advantages. First, it may significantly reduce the overall analysis cost both in time and space, as e.g. different computers can be used for the analysis of the context and the object. Second, as the total memory consumption is reduced, very precise analyses can be used for the context and/or the object. Third, it allows a form of modular analysis: if $\mathtt{o}$ is replaced by another object $\mathtt{o}'$ then the analysis of $\mathtt{C}[\mathtt{o}']$ requires just the analysis of $\mathtt{o}'$. For instance, this is the case when $\mathtt{C}[\cdot]$ is a function and $\mathtt{o}$ and $\mathtt{o}'$ are actual parameters, or when $\mathtt{o}'$ is a refinement of $\mathtt{o}$, e.g. $\mathtt{o}'$ is a sub-object of $\mathtt{o}$.

We present a generic *monolithic* static analysis of the context and the object $\bar{\mathbb{K}}[\![\mathtt{C}[\mathtt{o}]]\!]$, parameterized by an abstract domain $\bar{\mathsf{D}}$. We show how it can be split into two semantic functions, Γ and Θ, corresponding respectively to the analysis of the context and the object. The fundamental idea is the use of regular expressions for approximating the interactions between the context and the object, so that we refine the abstract domain $\bar{\mathsf{D}}$ with a domain of regular expressions. We have that:

- the object analysis Θ is a function that takes as input a map from objects to regular expressions. It returns a map from objects to their approximations.

- the context analysis Γ is a function that takes as input the approximation of the semantics of the objects. It returns an abstract value and a map from objects to regular expressions.

The functions Θ and Γ are mutually recursive. Thus, we handle this situation with the usual iterative approach. In particular, we begin by assuming the worst-case for the objects approximations and the contexts. Then, we show that the iterations form a chain of increasing precision, each step being a sound upper-approximation of $\bar{\mathbb{K}}[\![\mathtt{C}[\mathtt{o}]]\!]$. This implies that the iterations can be stopped as soon as the desired degree of precision is reached, enabling a trade-off between precision and cost.

10.2 Context Syntax and Semantics

We begin by defining the syntax and the semantics of a context. The goal is to study the interactions between the context and the objects it interacts with.

In order to simplify the notation we assume that it exists just one class and that the semantics of the context, of the methods and of the class constructor is deterministic. Nevertheless, the generalization of the results to the case of an arbitrary number of classes and to non-deterministic semantic is straightforward.

10.2.1 Syntax

The syntax of a context is quite standard, except that we distinguish three kinds of assignments: the assignment of the value of a side-effects free expression to a variable, the assignment of a value of one variable to another one and the assignment of the return value of a method call to a variable:

DEFINITION 10.1 (CONTEXT SYNTAX) *Let* $\mathtt{x}$, $\mathtt{x}_1$, $\mathtt{x}_2$ *and* $\mathtt{o}$ *be variables, let* $\mathtt{e}$ *be an expression and let* $\mathtt{b}$ *be a boolean expression. Then a context belongs to the language generated by the following grammar:*

$$\begin{aligned}\mathtt{C} ::= \ & \mathtt{A\ o\ =\ new\ A(E)} \mid \mathtt{C_1; C_2} \mid \mathtt{skip} \mid \mathtt{x = E} \mid \mathtt{x_1 = x_2} \\ & \mid \mathtt{x = o.m(E)} \mid \mathtt{if\ b\ then\ C_1\ else\ C_2} \mid \mathtt{while\ b\ do\ C}.\end{aligned}$$

$\mathtt{C}$ denotes an arbitrary context, $\mathtt{C}[\cdot]$ denotes a context that may contain one or more objects and $\mathtt{C}[\mathtt{o}]$ denotes a context that uses an object $\mathtt{o}$. However, as we allow aliasing of objects, we cannot give the formal definition of $\mathtt{C}[\mathtt{o}]$ on a strictly syntactic basis. Therefore, such a definition is postponed to the next section.

It is worth noting that in the previous definition we implicitly assume that a method call always returns a value. If this is not the case, we can suppose that it returns a `void` value that is assigned to some dummy variable.

10.2.2 Semantics

We define the context semantics in denotational style, by induction on the syntax. The semantics of expressions and that of the boolean expressions are assumed to be side-effect free, such that $\mathbb{e}[\![\mathtt{E}]\!] \in [\mathsf{Env} \times \mathsf{Store} \to \mathsf{Val}]$ and $\mathbb{b}[\![\mathtt{b}]\!] \in [\mathsf{Env} \times \mathsf{Store} \to \mathbb{B}]$. The semantics of the context is given by induction on the syntax:

$$
\begin{aligned}
\mathbb{k}[\![\texttt{A o } = \texttt{ new A(E)}]\!] = \lambda \mathsf{e},\mathsf{s}.\mathit{let}\ v = \mathbb{e}[\![\texttt{E}]\!](\mathsf{e},\mathsf{s}), a = \mathsf{alloc}(\mathsf{s}),\\
(\mathsf{e}_0,\mathsf{s}_0) = \mathbb{i}[\![\texttt{init}]\!](v,\mathsf{s}),\\
\mathsf{e}' = \mathsf{e}[\mathsf{o} \mapsto a], \mathsf{s}' = \mathsf{s}_0[a \mapsto \mathsf{e}_0]\\
\mathit{in}\ (\mathsf{e}',\mathsf{s}')\\
\mathbb{k}[\![\texttt{C}_1;\texttt{C}_2]\!] = \lambda \mathsf{e},\mathsf{s}.\mathbb{k}[\![\texttt{C}_2]\!](\mathbb{k}[\![\texttt{C}_1]\!](\mathsf{e},\mathsf{s})) \qquad \mathbb{k}[\![\texttt{skip}]\!] = \lambda \mathsf{e},\mathsf{s}.(\mathsf{e},\mathsf{s})\\
\mathbb{k}[\![\texttt{x} = \texttt{E}]\!] = \lambda \mathsf{e},\mathsf{s}.(\mathsf{e},\mathsf{s}[\mathsf{e}(\texttt{x}) \mapsto \mathbb{e}[\![\texttt{E}]\!](\mathsf{e},\mathsf{s})])\\
\mathbb{k}[\![\texttt{x}_1 = \texttt{x}_2]\!] = \lambda \mathsf{e},\mathsf{s}.(\mathsf{e}[\texttt{x}_1 \mapsto \mathsf{e}(\texttt{x}_2)],\mathsf{s})\\
\mathbb{k}[\![\texttt{x} = \texttt{o.m(E)}]\!] = \lambda \mathsf{e},\mathsf{s}.\mathit{let}\ v = \mathbb{e}[\![\texttt{E}]\!](\mathsf{e},\mathsf{s}),\\
(v_0,\mathsf{e}_0,\mathsf{s}_0) = \mathbb{m}[\![\texttt{m}]\!](v,\mathsf{s}(\mathsf{e}(\texttt{o})),\mathsf{s}),\\
\mathsf{s}' = \mathsf{s}_0[\mathsf{e}(\texttt{x}) \mapsto v_0, \mathsf{e}(\texttt{o}) \mapsto \mathsf{e}_0]\\
\mathit{in}\ (\mathsf{e},\mathsf{s}')\\
\mathbb{k}[\![\texttt{if b then C}_1\texttt{ else C}_2]\!] = \lambda \mathsf{e},\mathsf{s}.\mathit{if}\ \mathbb{b}[\![\texttt{b}]\!](\mathsf{e},\mathsf{s}) = \texttt{tt}\ \mathit{then}\ \mathbb{k}[\![\texttt{C}_1]\!](\mathsf{e},\mathsf{s})\ \mathit{else}\ \mathbb{k}[\![\texttt{C}_2]\!](\mathsf{e},\mathsf{s})\\
\mathbb{k}[\![\texttt{while b do C}]\!] = \mathrm{lfp}\lambda\phi.\lambda \mathsf{e},\mathsf{s}.\mathit{if}\ \mathbb{b}[\![\texttt{b}]\!](\mathsf{e},\mathsf{s}) = \texttt{tt}\ \mathit{then}\ \phi(\mathbb{k}[\![\texttt{C}]\!](\mathsf{e},\mathsf{s}))\ \mathit{else}\ (\mathsf{e},\mathsf{s})
\end{aligned}
$$

Figure 10.1: Semantics of the context

Definition 10.2 (Context Semantics, $\mathbb{k}[\![\cdot]\!]$) *Let* $\mathsf{alloc} \in [\mathsf{Store} \to \mathsf{Addr}]$ *be the function that returns a fresh memory address. Then, the semantics of a context,* $\mathbb{k}[\![\texttt{C}]\!] \in [\mathsf{Env} \times \mathsf{Store} \to \mathsf{Env} \times \mathsf{Store}]$ *is given in Figure 10.1.*

When a class `A` is instantiated, the initial value is evaluated, and the class constructor is invoked with that value and the store. The class constructor returns the environment e_0 of the new object and the modified store s_0. Then the environment and the store change, so that `o` points to the memory allocated for storing e_0. When a method of the object `o` is invoked, its environment is fetched from the memory and passed to the method. This implies that the method has no access the caller environment, but only to that of the object it belongs to. In other words, the context has the burden of setting the right environment for a method call, so that the handling of `this` is somehow transparent to the callee. For the rest, the semantics in Figure 10.1 is a quite standard denotational semantics [98]. In particular, the loop semantics is handled by the least fixpoint operator in the flat Scott-domain $\mathsf{Env} \times \mathsf{Store} \cup \{\bot\}$.

The identity of an object is given by the memory address where its environment is stored. Thus the concept of a context that can access an object is a semantic rather than a syntactic concept. So, we can formally define the writing `C`[`o`], i.e. the context `C` that uses an object `o`:

DEFINITION 10.3 (CONTEXT, $\mathtt{C}[\cdot]$) *Given a pair* $(\mathsf{e}_0, \mathsf{s}_0) \in \mathsf{Env} \times \mathsf{Store}$*, a context* $\mathtt{C}$ *uses an object* o *if* $\Bbbk[\![\mathtt{C}]\!](\mathsf{e}_0, \mathsf{s}_0) = (\mathsf{e}, \mathsf{s})$ *and*

$$\exists \mathtt{x} \in \mathtt{Var}.\mathsf{e}(\mathtt{x}) = \mathsf{o} \wedge \mathsf{s}(\mathsf{e}(\mathtt{x})) \in \mathsf{Env}.$$

In that case, we write $\mathtt{C}[\mathsf{o}]$*.*

10.2.3 Collecting Semantics

A semantic property is a set of possible semantics of a program. The set of semantic properties $\mathcal{P}(\mathsf{Env} \times \mathsf{Store})$ is a complete boolean lattice

$$\langle \mathcal{P}(\mathsf{Env} \times \mathsf{Store}), \subseteq, \emptyset, \mathsf{Env} \times \mathsf{Store}, \cup, \cap \rangle$$

for subset inclusion, that is logical implication. The standard collecting semantics of a program, $\mathbb{K}[\![\mathtt{C}]\!](\mathit{In}) = \{\Bbbk[\![\mathtt{C}]\!](\mathsf{e}, \mathsf{s}) \mid (\mathsf{e}, \mathsf{s}) \in \mathit{In}\}$, is the strongest program property. The goal of a static analysis is to find a computable approximation of $\mathbb{K}[\![\mathtt{C}]\!]$.

10.3 Monolithic Abstract Semantics

We proceed to the definition of a generic abstract semantics for the language presented in the previous section. First we consider the abstract semantic domains. Afterward, we present the abstract semantics for the class constructor and methods, and for the context.

10.3.1 Abstract Semantic Domains

The values in $\mathcal{P}(\mathsf{Val})$ are approximated by an abstract domain $\overline{\mathsf{Val}}$. The correspondence between the two domains is given by the Galois connection:

$$\langle \mathcal{P}(\mathsf{Val}), \subseteq, \emptyset, \mathsf{Val}, \cup, \cap \rangle \underset{\alpha_v}{\overset{\gamma_v}{\leftrightarrows}} \langle \overline{\mathsf{Val}}, \bar{\sqsubseteq}_v, \bar{\bot}_v, \bar{\top}_v, \bar{\sqcup}_v, \bar{\sqcap}_v \rangle.$$

The set of abstract addresses is $\overline{\mathsf{Addr}} \subseteq \overline{\mathsf{Val}}$. We assume $\overline{\mathsf{Addr}}$ to be a sublattice of $\overline{\mathsf{Val}}$. If $\mathsf{o} \in \mathsf{Addr}$ denotes an object in the concrete, then $\vartheta = \alpha_v(\{\mathsf{o}\})$ is the corresponding abstract address. On the other side, ϑ stands for the set of concrete addresses $\gamma_v(\vartheta)$, which may contain several objects. Therefore, ϑ approximates all the objects in $\gamma_v(\vartheta)$. With an abuse of language, we call ϑ an abstract object.

The domain $\bar{\mathsf{D}}$ abstracts the domain of concrete properties $\mathcal{P}(\mathsf{Env} \times \mathsf{Store})$ by means of a Galois connection:

$$\langle \mathcal{P}(\mathsf{Env} \times \mathsf{Store}), \subseteq, \emptyset, \mathsf{Env} \times \mathsf{Store}, \cup, \cap \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \bar{\mathsf{D}}, \bar{\sqsubseteq}, \bar{\bot}, \bar{\top}, \bar{\sqcup}, \bar{\sqcap} \rangle.$$

We call an element of $\bar{\mathsf{D}}$ an abstract state. In general, the domain $\bar{\mathsf{D}}$ is a relational abstraction of $\mathcal{P}(\mathsf{Env} \times \mathsf{Store})$. We consider two projections such that for each $\bar{\mathsf{d}} \in \bar{\mathsf{D}}$, $\pi_{\mathsf{e}}(\bar{\mathsf{d}})$ and $\pi_{\mathsf{s}}(\bar{\mathsf{d}})$ are, respectively, the projections of $\bar{\mathsf{d}}$ on the environment and the store. We use the brackets $\lfloor \cdot \rfloor$ to denote the inverse operation of the projection, i.e. given an abstraction for the environment and the store it returns the abstract state. Moreover, some operations are defined on $\bar{\mathsf{D}}$: $\overline{\mathsf{alloc}}$, $\overline{\mathsf{assign}}$, $\overline{\mathsf{true}}$ and $\overline{\mathsf{false}}$. The first one, $\overline{\mathsf{alloc}} \in [\bar{\mathsf{D}} \to \overline{\mathsf{Addr}}]$, is the abstract counterpart for memory allocation. It takes an approximation of the state and it returns an abstract address where the object environment can be stored. It satisfies the soundness requirement:

$$\forall \bar{\mathsf{d}} \in \bar{\mathsf{D}}.\{\mathsf{alloc}(\mathsf{s}) \mid (\mathsf{e},\mathsf{s}) \in \gamma(\bar{\mathsf{d}})\} \subseteq \gamma_v(\overline{\mathsf{alloc}}(\bar{\mathsf{d}})).$$

The function $\overline{\mathsf{assign}} \in [\bar{\mathsf{D}} \times (\mathrm{Var} \times \bar{\mathsf{D}})^+ \to \bar{\mathsf{D}}]$ handles the assignment in the abstract domain. It takes as input an abstract state and a non-empty list of bindings from variables to values. It returns the new abstract state. With an abuse of notation, we sometimes write $\overline{\mathsf{assign}}(\bar{\mathsf{d}}, \pi_{\mathsf{s}}(\bar{\mathsf{d}}) \mapsto \pi_{\mathsf{s}}(\bar{\mathsf{d}}_0))$ to denote that the abstract store $\pi_{\mathsf{s}}(\bar{\mathsf{d}})$ is updated by $\pi_{\mathsf{s}}(\bar{\mathsf{d}}_0)$.

Moreover, $\overline{\mathsf{true}}$, $\overline{\mathsf{false}} \in [\mathrm{BExp} \times \bar{\mathsf{D}} \to \bar{\mathsf{D}}]$ are the functions that given a boolean expression and an abstract element $\bar{\mathsf{d}}$ return an abstraction of the pairs $(\mathsf{e},\mathsf{s}) \in \gamma(\bar{\mathsf{d}})$ that make the condition respectively true or false. For instance $\overline{\mathsf{true}}$ is such that:

$$\forall \mathtt{b} \in \mathrm{BExp}.\forall \bar{\mathsf{d}} \in \bar{\mathsf{D}}.\{(\mathsf{e},\mathsf{s}) \mid \mathbb{b}[\![\mathtt{b}]\!](\mathsf{e},\mathsf{s}) = \mathsf{tt}\} \cap \gamma(\bar{\mathsf{d}}) \subseteq \overline{\mathsf{true}}(\mathtt{b},\bar{\mathsf{d}}).$$

10.3.2 Abstract Object Semantics

The abstract semantics for the class constructor and methods we consider here is slightly different from that considered in Chapter 5. The abstract counterpart for the constructor semantics is a function $\bar{\mathbb{I}}[\![\mathtt{init}]\!] \in [\overline{\mathsf{Val}} \times \bar{\mathsf{D}} \to \bar{\mathsf{D}}]$, which takes an abstract value and an abstract state and returns an abstract environment, that of the new object, and an abstract store. The abstract semantics for methods is similar. It is a function $\bar{\mathbb{M}}[\![\mathtt{m}]\!] \in [\overline{\mathsf{Val}} \times \bar{\mathsf{D}} \to \overline{\mathsf{Val}} \times \bar{\mathsf{D}}]$. The input is an abstract value and an abstract state, and the output is an abstraction of the return value and a modified abstract state.

10.3.3 Monolithic Abstract Context Semantics

The abstract semantics for contexts is defined on the top of the abstract semantics for the expressions and the basic operations of the abstract domain $\bar{\mathsf{D}}$. In particular, the abstract semantics of expressions is $\bar{\mathbb{e}}[\![\mathtt{E}]\!] \in [\bar{\mathsf{D}} \to \overline{\mathsf{Val}}]$. It must satisfy the soundness requirement:

$$\forall (\mathsf{e},\mathsf{s}) \in \mathsf{Env} \times \mathsf{Store}.\ \mathbb{e}[\![\mathtt{E}]\!](\mathsf{e},\mathsf{s}) \in \gamma_v \circ \bar{\mathbb{e}}[\![\mathtt{E}]\!] \circ \alpha(\{(\mathsf{e},\mathsf{s})\}).$$

$$
\begin{aligned}
\bar{\mathbb{K}}[\![\texttt{A o } = \texttt{ new A(E)}]\!] &= \lambda\bar{\mathsf{d}}.\mathit{let}\ \bar{\mathsf{v}} = \bar{\mathbb{e}}[\![\texttt{E}]\!](\bar{\mathsf{d}}), \vartheta = \overline{\mathsf{alloc}}(\bar{\mathsf{d}}),\\
&\qquad \bar{\mathsf{d}}_0 = \bar{\mathbb{I}}[\![\texttt{init}]\!](\bar{\mathsf{v}}, \bar{\mathsf{d}})\\
&\qquad \mathit{in}\ \overline{\mathsf{assign}}(\bar{\mathsf{d}}, \vartheta \mapsto \pi_{\mathsf{e}}(\bar{\mathsf{d}}_0), \pi_{\mathsf{s}}(\bar{\mathsf{d}}) \mapsto \pi_{\mathsf{s}}(\bar{\mathsf{d}}_0))\\
\bar{\mathbb{K}}[\![\texttt{C}_1\texttt{;}\,\texttt{C}_2]\!] &= \lambda\bar{\mathsf{d}}.\bar{\mathbb{K}}[\![\texttt{C}_2]\!](\bar{\mathbb{K}}[\![\texttt{C}_1]\!](\bar{\mathsf{d}})) \qquad \bar{\mathbb{K}}[\![\texttt{skip}]\!] = \lambda\bar{\mathsf{d}}.\bar{\mathsf{d}}\\
\bar{\mathbb{K}}[\![\texttt{x} = \texttt{E}]\!] &= \lambda\bar{\mathsf{d}}.\overline{\mathsf{assign}}(\bar{\mathsf{d}}, \texttt{x} \mapsto \bar{\mathbb{e}}[\![\texttt{E}]\!](\bar{\mathsf{d}}))\\
\bar{\mathbb{K}}[\![\texttt{x}_1 = \texttt{x}_2]\!] &= \lambda\bar{\mathsf{d}}.\overline{\mathsf{assign}}(\bar{\mathsf{d}}, \texttt{x}_1 \mapsto \pi_{\mathsf{e}}(\bar{\mathsf{d}})(\texttt{x}_2))\\
\bar{\mathbb{K}}[\![\texttt{x} = \texttt{o.m(E)}]\!] &= \lambda\bar{\mathsf{d}}.\mathit{let}\ \bar{\mathsf{v}} = \bar{\mathbb{e}}[\![\texttt{E}]\!](\bar{\mathsf{d}}), \vartheta = \pi_{\mathsf{e}}(\bar{\mathsf{d}})(\texttt{o}),\\
&\qquad (\bar{\mathsf{v}}_0, \bar{\mathsf{d}}_0) = \bar{\mathbb{M}}[\![\texttt{m}]\!](\bar{\mathsf{v}}, \lfloor \pi_{\mathsf{s}}(\bar{\mathsf{d}})(\vartheta), \pi_{\mathsf{s}}(\bar{\mathsf{d}})\rfloor),\\
&\qquad \mathit{in}\ \overline{\mathsf{assign}}(\bar{\mathsf{d}}, \texttt{x} \mapsto \bar{\mathsf{v}}_0, \vartheta \mapsto \pi_{\mathsf{e}}(\bar{\mathsf{d}}_0), \pi_{\mathsf{s}}(\bar{\mathsf{d}}) \mapsto \pi_{\mathsf{s}}(\bar{\mathsf{d}}_0))\\
\bar{\mathbb{K}}[\![\texttt{if b then C}_1\texttt{ else C}_2]\!] &= \lambda\bar{\mathsf{d}}.\bar{\mathbb{K}}[\![\texttt{C}_1]\!](\overline{\mathsf{true}}(\texttt{b}, \bar{\mathsf{d}}))\bar{\sqcup}\bar{\mathbb{K}}[\![\texttt{C}_2]\!](\overline{\mathsf{false}}(\texttt{b}, \bar{\mathsf{d}}))\\
\bar{\mathbb{K}}[\![\texttt{while b do C}]\!] &= \lambda\bar{\mathsf{d}}.\overline{\mathsf{false}}(\texttt{b}, \mathrm{lfp}^{\bar{\sqsubseteq}}_{\bar{\mathsf{d}}}\lambda x.\bar{\mathbb{K}}[\![\texttt{C}]\!](\overline{\mathsf{true}}(\texttt{b}, x)))
\end{aligned}
$$

Figure 10.2: Monolithic abstract semantics

The generic abstract semantics mimics the concrete semantics. In particular, when a method $\texttt{m}$ is invoked, the corresponding abstract function $\bar{\mathbb{M}}[\![\texttt{m}]\!]$ is used. In practice, this means that the body of a method $\texttt{m}$ is analyzed from scratch at each invocation. Therefore the encapsulation of the object w.r.t. context is not exploited in the analysis. We call such an abstract semantics a *monolithic* abstract semantics in order to differentiate it from the separate compositional abstract semantics that we will introduce in the next section.

Definition 10.4 (Monolithic Abstract Context Semantics, $\bar{\mathbb{K}}[\![\cdot]\!]$) *The monolithic abstract context semantics* $\bar{\mathbb{K}}[\![\texttt{C}]\!] \in [\bar{\mathsf{D}} \to \bar{\mathsf{D}}]$ *is defined in Figure 10.2.*

The semantics in Figure 10.2 is quite similar to the concrete one in Figure 10.1. It is worth noting that the burden of handling the assignment is left to the underlying abstract domain $\bar{\mathsf{D}}$, and in particular to the function $\overline{\mathsf{assign}}$.

In general the abstract domains $\overline{\mathsf{Val}}$ and $\bar{\mathsf{D}}$ may not respect the Ascending Chain Condition (ACC) [88], so that the convergence of the analysis is enforced through the widening operators $\bar{\nabla}_v \in [\overline{\mathsf{Val}} \times \overline{\mathsf{Val}} \to \overline{\mathsf{Val}}]$ and $\bar{\nabla} \in [\bar{\mathsf{D}} \times \bar{\mathsf{D}} \to \bar{\mathsf{D}}]$.

The soundness of the above semantics is a consequence of the definitions of this section:

THEOREM 10.1 (SOUNDNESS OF THE MONOLITHIC ABSTRACT SEMANTICS) *The monolithic context abstract semantics is a sound approximation of the concrete semantics:*

$$\forall In \in \mathcal{P}(\mathsf{Env} \times \mathsf{Store}).\mathbb{K}[\![\mathtt{C}]\!](In) \subseteq \gamma \circ \bar{\mathbb{K}}[\![\mathtt{C}]\!] \circ \alpha(In).$$

Proof. (Sketch) We consider just a didactic case of the method invocation. By theorem hypotheses, we have that $\bar{\mathsf{v}}$ is a sound approximation of $\mathbb{e}[\![\mathtt{E}]\!](\gamma(\bar{\mathsf{d}}))$, $\gamma(\vartheta) \ni \pi_{\mathsf{e}}(\gamma(\bar{\mathsf{d}}))(o)$. Thus, as $\bar{\mathbb{M}}[\![\mathtt{m}]\!]$ is a sound abstraction of the method collecting semantics then $(\bar{\mathsf{v}}_0, \bar{\mathsf{d}}_0)$ is a sound approximation of the method invocation. The proof of such a case follows by the soundness of $\overline{\mathsf{assign}}$.

The full proof of the theorem proceeds by structural induction on the syntax of the contexts and by induction on the iterations for the `while`.

q.e.d.

10.4 Separate Abstract Semantics

The abstract semantics $\bar{\mathbb{K}}[\![\cdot]\!]$ defined in the previous section does not take into account the encapsulation features of object-oriented languages, so that, for instance each time a method of an object is invoked, its body must be analyzed. In this section we show how to split $\bar{\mathbb{K}}[\![\cdot]\!]$ into two parts. The first part analyzes the context using an approximation of the object. The latter analyzes the object using an approximation of the context. This reduces the overall cost of the analysis, as it can be parallelized.

10.4.1 Regular Expressions Domain

The main idea for the separate analysis is to refine the abstract domain $\bar{\mathsf{D}}$ with the abstract domain R of regular expressions over the infinite alphabet $(\{\mathtt{init}\} \cup \mathcal{P}(\mathtt{M})) \times \overline{\mathsf{Val}} \times \bar{\mathsf{D}}$. Given an object, the intuition behind the refinement is to use a regular expression to abstract the method's invocations performed by the context. In particular, each *letter* in the alphabet represents a set of methods that can be invoked, an approximation of their input values and an approximation of the state. Such a regular expression is built during the analysis of the context. Then it is used for the analysis of the object.

The definition of the regular expressions in R is given by structural induction. The base cases are the *null* string ε and the letters l of the alphabet $(\{\mathtt{init}\} \cup \mathcal{P}(\mathtt{M})) \times \overline{\mathsf{Val}} \times \bar{\mathsf{D}}$. Then, if r_1 and r_2 are regular expressions so are the concatenation $r_1 \cdot r_2$, the union $r_1 + r_2$ and the Kleene-closure r_1^*.

$$
\begin{aligned}
\top_r \nabla_r x &= x \nabla_r \top_r = \top_r & x \nabla_r \varepsilon &= \varepsilon \nabla_r x = x \\
\langle \mathtt{m}, \bar{\mathtt{v}}, \bar{\mathtt{s}} \rangle \nabla_r \langle \mathtt{m}_1, \bar{\mathtt{v}}_1, \bar{\mathtt{s}}_1 \rangle &= \langle \mathtt{m} \cup \mathtt{m}_1, \bar{\mathtt{v}} \bar{\nabla}_v \bar{\mathtt{v}}_1, \bar{\mathtt{s}} \bar{\nabla} \bar{\mathtt{s}}_1 \rangle & (r_1 \cdot r_2) \nabla_r n &= (r_1 \nabla_r n) \cdot r_2 \\
(r_1 + r_2) \nabla_r n &= (r_1 \nabla_r n) + (r_2 \nabla_r n) & r^* \nabla_r n &= (r \nabla_r n)^* \\
(r_1 \cdot r_2) \nabla_r (r_1' \cdot r_2') &= (r_1 \nabla_r r_1') \cdot (r_2 \nabla_r r_2') & r_1^* \nabla_r r_2^* &= (r_1 \nabla_r r_2)^* \\
(r_1 + r_2) \nabla_r (r_1' + r_2') &= (r_1 \nabla_r r_1') + (r_2 \nabla_r r_2') \\
x \nabla_r y &= \top_r \qquad \text{in all the other cases}
\end{aligned}
$$

Figure 10.3: Widening on regular expressions

Definition 10.5 (Regular Expression) *Given a class* $\mathtt{A} = \langle \mathtt{init}, \mathtt{F}, \mathtt{M} \rangle$ *the abstract domains* $\bar{\mathsf{D}}$ *and* $\overline{\mathsf{Val}}$*, the set of regular expressions* R *is the language generated by the following grammar:*

$$
\begin{aligned}
r &::= \varepsilon \mid l \mid r_1 \cdot r_2 \mid r_1 + r_2 \mid r^* \\
l &\in (\{\mathtt{init}\} \cup \mathcal{P}(\mathtt{M})) \times \overline{\mathsf{Val}} \times \bar{\mathsf{D}}.
\end{aligned}
$$

The language generated by a regular expression is defined by structural induction:

Definition 10.6 (Language generated by a regular expression) *The language generated by a regular expression* r *is:*

$$
\begin{aligned}
\mathcal{L}(\langle \mathtt{ms}, \bar{\mathtt{v}}, \bar{\mathtt{s}} \rangle) &= \{\langle \mathtt{m}, v, s \rangle \mid \mathtt{m} \in \mathtt{ms}, v \in \gamma_v(\bar{\mathtt{v}}), s \in \gamma(\bar{\mathtt{s}})\} \qquad \mathcal{L}(\varepsilon) = \emptyset \\
\mathcal{L}(r_1 \cdot r_2) &= \{s_1 \cdot s_2 \mid s_1 \in \mathcal{L}(r_1), s_2 \in \mathcal{L}(r_2)\} \\
\mathcal{L}(r_1 + r_2) &= \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r^*) &= \mathrm{lfp}_{\emptyset}^{\subseteq} \lambda X. \mathcal{L}(r) \cup \{s_1 \cdot s_2 \mid s_1 \in X, s_2 \in \mathcal{L}(r)\}.
\end{aligned}
$$

The order on regular expressions is a direct consequence of the above definition:

$$\forall r_1, r_2 \in \mathsf{R}. r_1 \sqsubseteq_r r_2 \iff \mathcal{L}(r_1) \subseteq \mathcal{L}(r_2).$$

So, two expressions are equivalent if they generate the same language: $r_1 \equiv r_2 \iff \mathcal{L}(r_1) = \mathcal{L}(r_2)$.

The expression $\langle \{\mathtt{init}\} \cup \mathtt{M}, \bar{\top}_v, \bar{\top} \rangle^* \in \mathsf{R}$ stands for a context that may invoke any method, with any input value and with any memory configuration for a non-specified number of times. So, it gives no information. Thus, the largest element is $\top_r = \langle \{\mathtt{init}\} \cup \mathtt{M}, \bar{\top}_v, \bar{\top} \rangle^*$.

The join of two regular expressions is simply their union: $\forall r_1, r_2 \in \mathsf{R}. r_1 \sqcup_r r_2 = r_1 + r_2$. Similarly, it is possible to define the meet operator $\sqcap_r$, so that

LEMMA 10.1 (LATTICE OF REGULAR EXPRESSIONS) $\langle \mathsf{R}, \sqsubseteq_r, \varepsilon, \top_r, \sqcup_r, \sqcap_r \rangle$ *is a complete lattice.*

Proof. We are left to show that the empty expression ε is the least element. We have that, by definition of $\mathcal{L}(\varepsilon) = \emptyset$, so that $\forall r \in \mathsf{R}.\ \varepsilon \sqsubseteq_r r$.

q.e.d.

It is immediate to see that the domain R does not satisfy the ACC, so we need the operator of Figure 10.3 to deal with strictly increasing chains of regular expressions. There are two intuitions behind the operator ∇_r. The first one is to preserve the syntactic structure of the regular expressions between two successive iterations, so that the number of $\{\cdot, +, ^*\}$ does not increase. The second one is to propagate the ∇_r inside the regular expressions in order to use the widenings on $\overline{\mathsf{Val}}$ and $\bar{\mathsf{D}}$. Convergence is assured as $\mathtt{M}$ is a finite set, and $\bar{\nabla}_v$ and $\bar{\nabla}$ are widenings on the respective domains.

LEMMA 10.2 (WIDENING OF REGULAR EXPRESSIONS) *The operator* $\nabla_r \in [\mathsf{R} \times \mathsf{R} \to \mathsf{R}]$ *defined in Figure 10.3 is a widening.*

10.4.2 Interaction History

For the purpose of our analysis, we need to associate with each abstract address, i.e. a set of concrete objects, a regular expression that denotes the interaction of the context on it. As a consequence we consider the functional lifting of the abstract domain of regular expressions, $\dot{\mathsf{R}} = [\overline{\mathsf{Addr}} \to \mathsf{R}]$. The order $\dot{\sqsubseteq}_r$ is defined pointwise:

$$\forall \dot{r}_1, \dot{r}_2 \in \dot{\mathsf{R}}. \dot{r}_1 \dot{\sqsubseteq}_r \dot{r}_2 \Leftrightarrow \forall \vartheta \in \overline{\mathsf{Addr}}. \dot{r}_1(\vartheta) \sqsubseteq_r \dot{r}_2(\vartheta).$$

Similarly, the join and meet are defined point-wise: $\forall \dot{r}_1, \dot{r}_2 \in \dot{\mathsf{R}}$.

$$\dot{r}_1 \dot{\sqcup}_r \dot{r}_2 = \lambda \vartheta.\ \dot{r}_1(\vartheta) \sqcup_r \dot{r}_2(\vartheta)$$
$$\dot{r}_1 \dot{\sqcap}_r \dot{r}_2 = \lambda \vartheta.\ \dot{r}_1(\vartheta) \sqcap_r \dot{r}_2(\vartheta)$$

so that, because of basic domain theory:

LEMMA 10.3 $\langle \dot{\mathsf{R}}, \dot{\sqsubseteq}_r, \lambda\vartheta.\varepsilon, \lambda\vartheta.\top_r, \dot{\sqcup}_r, \dot{\sqcap}_r \rangle$ *is a complete lattice.*

We call an element $\dot{r} \in \dot{\mathsf{R}}$ an interaction history.

10.4.3 Separate Object Analysis

The goal of the separate object analysis is to infer an object invariant and the method postconditions when the instantiation context is approximated by a regular expression. Thus, the input of the abstract semantics $\bar{\mathbb{O}}[\![\vartheta]\!]$ is a regular expression r and an initial abstract value for the object fields and the method preconditions. The output is an invariant for the object fields and the method postconditions, under the context represented by r. A postcondition is a pair consisting of an approximation of the return value and an abstract state. Thus, the result is an element of the abstract domain $\bar{\mathsf{O}} = \bar{\mathsf{D}} \times [\mathtt{M} \to \overline{\mathsf{Val}} \times \bar{\mathsf{D}}]$. From basic domain theory [98], the orders on $\bar{\mathsf{D}}$ and $\overline{\mathsf{Val}}$ induce the order on $\bar{\mathsf{O}}$. So, for instance the order is

$$\bar{\sqsubseteq}_o = \bar{\sqsubseteq} \times (\dot{\bar{\sqsubseteq}}_v \times \bar{\sqsubseteq}),$$

the least element is $\bar{\bot}_o = \langle \bar{\bot}, \lambda \mathtt{m}.\langle \bar{\bot}_v, \bar{\bot} \rangle \rangle$ and the largest is $\bar{\top}_o = \langle \bar{\top}, \lambda \mathtt{m}.\ \langle \bar{\top}_v, \bar{\top} \rangle \rangle$. The meet, the join and the widening can be defined in a similar fashion, so that

LEMMA 10.4 *$\langle \bar{\mathsf{O}}, \bar{\sqsubseteq}_o, \bar{\bot}_o, \bar{\top}_o, \bar{\sqcup}_o, \bar{\sqcap}_o \rangle$ is a complete lattice.*

The definition of $\bar{\mathbb{O}}[\![\vartheta]\!]$ is by structural induction on the regular expressions:

DEFINITION 10.7 (SEPARATE OBJECT ANALYSIS, $\bar{\mathbb{O}}[\![\cdot]\!]$) *The separate object abstract semantics $\bar{\mathbb{O}}[\![\vartheta]\!] \in [\mathsf{R} \times \bar{\mathsf{O}} \to \bar{\mathsf{O}}]$ is defined in Figure 10.4.*

The base cases are the empty expression ε and the letters $\langle \mathtt{ms}, \bar{\mathsf{v}}, \bar{\mathsf{s}} \rangle$ and $\langle \{\mathtt{init}\}, \bar{\mathsf{v}}, \bar{\mathsf{s}} \rangle$. In the first case the context does not perform any action, so that the state of the object does not change at all. In the latter, the context may invoke any method $\mathtt{m_i} \in \mathtt{ms}$. The abstract value $\bar{\bigsqcup}\bar{\mathsf{s}}_\mathsf{i}$ approximates the object field values after calling the method $\mathtt{m}_1$ *or* $\mathtt{m}_2$ *or* $\ldots$ *or* $\mathtt{m}_\mathtt{n}$. As a consequence, $\bar{\mathsf{i}} \bar{\sqcup} \bar{\bigsqcup} \bar{\mathsf{s}}_\mathsf{i}$ approximates the object fields before and after executing any method in $\mathtt{ms}$. Hence, it is an object invariant. On the other hand, if $\langle \bar{\mathsf{w}}_i, \bar{\mathsf{q}}_i \rangle$ is the initial approximation of the return values and the states reached after the execution of a method $\mathtt{m_i} \in \mathtt{ms}$, then $\langle \bar{\mathsf{w}}_i \bar{\sqcup}_v \bar{\mathsf{v}}_i, \bar{\mathsf{q}}_i \bar{\sqcup} \bar{\mathsf{s}}_i \rangle$ is the postcondition of $\mathtt{m_i}$ after its execution. The case of the constructor `init` is quite similar.

As for the inductive cases are concerned, the rules for concatenation and union formalize respectively that *"the context first performs r_1 and then r_2"* and *"the context can perform either r_1 or r_2"*. Finally, the rule for the Kleene-closure is a little bit more tricky. In fact the intuitive meaning of r^* is that, starting from an initial abstract value $\langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle$ the context performs the

$$
\begin{aligned}
\bar{\mathbb{O}}[\![\vartheta]\!](\varepsilon, \langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle) =& \langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle \\
\bar{\mathbb{O}}[\![\vartheta]\!](\langle \{\mathtt{init}\}, \bar{\mathsf{v}}, \bar{\mathsf{s}} \rangle, \langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle) =& \mathit{let}\ \langle \bar{\mathsf{e}}_0, \bar{\mathsf{s}}_0 \rangle = \bar{\mathbb{I}}[\![\mathtt{init}]\!](\bar{\mathsf{v}}, \bar{\mathsf{s}} \bar{\sqcup} \bar{\mathsf{i}}) \\
& \mathit{in}\ \langle \bar{\mathsf{i}} \bar{\sqcup} \bar{\mathsf{s}}_0, \bar{\mathsf{p}}[\mathtt{init} \mapsto \langle \bar{\bot}_v, \bar{\mathsf{e}}_0 \rangle] \rangle \\
\bar{\mathbb{O}}[\![\vartheta]\!](\langle \mathtt{ms}, \bar{\mathsf{v}}, \bar{\mathsf{s}} \rangle, \langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle) =& \mathit{let}\ \forall \mathtt{m_i} \in \mathtt{ms}.(\bar{\mathsf{v}}_i, \bar{\mathsf{s}}_i) = \bar{\mathbb{M}}[\![\mathtt{m_i}]\!](\bar{\mathsf{v}}, \bar{\mathsf{s}} \bar{\sqcup} \bar{\mathsf{i}}), \\
& \langle \bar{\mathsf{w}}_i, \bar{\mathsf{q}}_i \rangle = \bar{\mathsf{p}}(\mathtt{m_i}) \\
& \mathit{in}\ (\bar{\mathsf{i}} \bar{\sqcup} \bar{\bigsqcup} \bar{\mathsf{s}}_i, \bar{\mathsf{p}}[\mathtt{m_i} \mapsto \langle \bar{\mathsf{w}}_i \bar{\sqcup}_v \bar{\mathsf{v}}_i, \bar{\mathsf{q}}_i \bar{\sqcup} \bar{\mathsf{s}}_i \rangle]) \\
\\
\bar{\mathbb{O}}[\![\vartheta]\!](r_1 \cdot r_2, \langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle) =& \mathit{let}\ (\bar{\mathsf{i}}_1, \bar{\mathsf{p}}_1) = \bar{\mathbb{O}}[\![\vartheta]\!](r_1, \langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle), \\
& (\bar{\mathsf{i}}_2, \bar{\mathsf{p}}_2) = \bar{\mathbb{O}}[\![\vartheta]\!](r_2, (\bar{\mathsf{i}}_1, \bar{\mathsf{p}}_1)) \\
& \mathit{in}\ (\bar{\mathsf{i}}, \bar{\mathsf{p}}) \bar{\sqcup}_o (\bar{\mathsf{i}}_1, \bar{\mathsf{p}}_1) \bar{\sqcup}_o (\bar{\mathsf{i}}_2, \bar{\mathsf{p}}_2) \\
\bar{\mathbb{O}}[\![\vartheta]\!](r_1 + r_2, \langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle) =& \mathit{let}\ (\bar{\mathsf{i}}_1, \bar{\mathsf{p}}_1) = \bar{\mathbb{O}}[\![\vartheta]\!](r_1, \langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle), \\
& (\bar{\mathsf{i}}_2, \bar{\mathsf{p}}_2) = \bar{\mathbb{O}}[\![\vartheta]\!](r_2, \langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle) \\
& \mathit{in}\ (\bar{\mathsf{i}}, \bar{\mathsf{p}}) \bar{\sqcup}_o (\bar{\mathsf{i}}_1, \bar{\mathsf{p}}_1) \bar{\sqcup}_o (\bar{\mathsf{i}}_2, \bar{\mathsf{p}}_2) \\
\bar{\mathbb{O}}[\![\vartheta]\!](r^*, \langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle) =& \mathrm{lfp}^{\bar{\sqsubseteq}_o}_{\langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle}\ \lambda x, y.\bar{\mathbb{O}}[\![\vartheta]\!](r, (x, y))
\end{aligned}
$$

Figure 10.4: Separate object abstract semantics

interaction encoded by r an unspecified number of times. We handle this case by considering the least fixpoint greater than $\langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle$ according to the order $\bar{\sqsubseteq}_o$ on $\bar{\mathsf{O}}$. Nevertheless, if the abstract domains $\bar{\mathsf{D}}$ and $\overline{\mathsf{Val}}$ do not respect the ACC then the convergence of the iteration must be enforced. In that case, we use the following pointwise widening operator to force the convergence:

$$\lambda(\bar{\mathsf{i}}, \bar{\mathsf{p}}), (\bar{\mathsf{i}}', \bar{\mathsf{p}}').(\bar{\mathsf{i}} \bar{\nabla} \bar{\mathsf{i}}', \lambda \mathtt{m}.\bar{\mathsf{p}}(\mathtt{m})\ (\bar{\nabla}_v \times \bar{\nabla})\ \bar{\mathsf{p}}'(\mathtt{m})).$$

The regular expression $r_\top = \langle \{\mathtt{init}\}, \bar{\top}_v, \bar{\top} \rangle \cdot \top_r$ stands for a context that calls at first the class constructor with an unknown value and then may invoke any object method, with any possible value, an unspecified number of times. Thus the abstract value $\langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle = \bar{\mathbb{O}}[\![\vartheta]\!](r_\top, \bar{\bot}_o)$ is such that $\bar{\mathsf{i}}$ is a property of the object fields valid for all the object instances, in any context. So it is a class invariant in the sense of Chapter 5.

10.4.4 Separate Context Analysis

The separate context analysis $\bar{\mathbb{S}}[\![\mathtt{C}[\cdot]]\!]$ has two goals. The first goal is to analyze $\mathtt{C}[\mathtt{o}]$ without referring to the $\mathtt{o}$ code, but just to a pre-computed approximation of its semantics. The second goal is to infer, for each object

$$
\begin{aligned}
\bar{\mathbb{S}}[\![\texttt{A o } = \texttt{ new A(E)}]\!] &= \lambda \bar{\mathsf{d}}, \dot{\vartheta}, \dot{r}.\mathit{let}\ \bar{\mathsf{v}} = \bar{\mathbb{e}}[\![\texttt{E}]\!](\bar{\mathsf{d}}), \vartheta = \overline{\mathsf{alloc}}(\bar{\mathsf{d}}),\\
&\qquad \langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle = \dot{\vartheta}(\vartheta),\ \langle \bar{\perp}_v, \bar{\mathsf{d}}_0 \rangle = \bar{\mathsf{p}}(\texttt{init}),\\
&\qquad \dot{r}' = \dot{r}[\vartheta \mapsto \langle \{\texttt{init}\}, \bar{\mathsf{v}}, \bar{\mathsf{d}} \rangle \sqcup_r \dot{r}(\vartheta)]\\
&\qquad \mathit{in}\ (\overline{\mathsf{assign}}(\bar{\mathsf{d}}, \vartheta \mapsto \pi_{\mathsf{e}}(\bar{\mathsf{d}}_0), \pi_{\mathsf{s}}(\bar{\mathsf{d}}) \mapsto \pi_{\mathsf{s}}(\bar{\mathsf{d}}_0)), \dot{r}')\\
\bar{\mathbb{S}}[\![\texttt{C}_1\texttt{; C}_2]\!] &= \lambda \bar{\mathsf{d}}, \dot{\vartheta}, \dot{r}.\mathit{let}\ (\bar{\mathsf{d}}_1, \dot{r}_1) = \bar{\mathbb{S}}[\![\texttt{C}_1]\!](\bar{\mathsf{d}}, \dot{\vartheta}, \dot{r})\\
&\qquad \mathit{in}\ \bar{\mathbb{S}}[\![\texttt{C}_2]\!](\bar{\mathsf{d}}_1, \dot{\vartheta}, \dot{r}_1)\\
\bar{\mathbb{S}}[\![\texttt{skip}]\!] &= \lambda \bar{\mathsf{d}}, \dot{\vartheta}, \dot{r}.(\bar{\mathsf{d}}, \dot{r})\\
\bar{\mathbb{S}}[\![\texttt{x} = \texttt{E}]\!] &= \lambda \bar{\mathsf{d}}, \dot{\vartheta}, \dot{r}.(\overline{\mathsf{assign}}(\bar{\mathsf{d}}, \texttt{x} \mapsto \bar{\mathbb{e}}[\![\texttt{E}]\!](\bar{\mathsf{d}})), \dot{r})\\
\bar{\mathbb{S}}[\![\texttt{x}_1 = \texttt{x}_2]\!] &= \lambda \bar{\mathsf{d}}, \dot{\vartheta}, \dot{r}.(\overline{\mathsf{assign}}(\bar{\mathsf{d}}, \texttt{x}_1 \mapsto \pi_{\mathsf{e}}(\bar{\mathsf{d}})(\texttt{x}_2)), \dot{r})\\
\bar{\mathbb{S}}[\![\texttt{x} = \texttt{o.m(E)}]\!] &= \lambda \bar{\mathsf{d}}, \dot{\vartheta}, \dot{r}.\mathit{let}\ \bar{\mathsf{v}} = \bar{\mathbb{e}}[\![\texttt{E}]\!](\bar{\mathsf{d}}), \vartheta = \pi_{\mathsf{e}}(\bar{\mathsf{d}})(\texttt{o}),\\
&\qquad \langle \bar{\mathsf{i}}, \bar{\mathsf{p}} \rangle = \dot{\vartheta}(\vartheta), \langle \bar{\mathsf{v}}_{\texttt{m}}, \bar{\mathsf{q}}_{\texttt{m}} \rangle = \bar{\mathsf{p}}(\texttt{m})\\
&\qquad \bar{\mathsf{d}}' = \overline{\mathsf{assign}}(\bar{\mathsf{d}}, \texttt{x} \mapsto \bar{\mathsf{v}}_{\texttt{m}}, \vartheta \mapsto \pi_{\mathsf{e}}(\bar{\mathsf{q}}_{\texttt{m}}), \pi_{\mathsf{s}}(\bar{\mathsf{d}}) \mapsto \pi_{\mathsf{s}}(\bar{\mathsf{q}}_{\texttt{m}})),\\
&\qquad \dot{r}' = \dot{r}[\vartheta \mapsto \dot{r}(\vartheta) \cdot \langle \texttt{m}, \bar{\mathsf{v}}, \lfloor \pi_{\mathsf{s}}(\bar{\mathsf{d}})(\vartheta), \pi_{\mathsf{s}}(\bar{\mathsf{d}}) \rfloor \rangle]\\
&\qquad \mathit{in}\ (\bar{\mathsf{d}}', \dot{r}')\\
\bar{\mathbb{S}}[\![\texttt{if b then C}_1\texttt{ else C}_2]\!] &= \lambda \bar{\mathsf{d}}, \dot{\vartheta}, \dot{r}.\mathit{let}\ (\bar{\mathsf{d}}_1, \dot{r}_1) = \bar{\mathbb{S}}[\![\texttt{C}_1]\!](\overline{\mathsf{true}}(\texttt{b}, \bar{\mathsf{d}}), \dot{\vartheta}, \dot{r}),\\
&\qquad (\bar{\mathsf{d}}_2, \dot{r}_2) = \bar{\mathbb{S}}[\![\texttt{C}_2]\!](\overline{\mathsf{false}}(\texttt{b}, \bar{\mathsf{d}}), \dot{\vartheta}, \dot{r})\\
&\qquad \mathit{in}\ (\bar{\mathsf{d}}_1 \bar{\sqcup} \bar{\mathsf{d}}_2, \dot{r}_1 \dot{\sqcup}_r \dot{r}_2)\\
\bar{\mathbb{S}}[\![\texttt{while b do C}]\!] &= \lambda \bar{\mathsf{d}}, \dot{\vartheta}, \dot{r}.\mathit{let}\ (\bar{\mathsf{d}}', \dot{r}') = \mathrm{lfp}^{\bar{\sqsubseteq} \times \dot{\sqsubseteq}_r}_{(\bar{\mathsf{d}}, \lambda\vartheta.\varepsilon)} \lambda(x, y).\bar{\mathbb{S}}[\![\texttt{C}]\!](\overline{\mathsf{true}}(\texttt{b}, x), \dot{\vartheta}, y)\\
&\qquad \mathit{in}\ (\overline{\mathsf{false}}(\texttt{b}, \bar{\mathsf{d}}'), \lambda\vartheta.\dot{r}(\vartheta) \cdot (\dot{r}'(\vartheta))^*)
\end{aligned}
$$

Figure 10.5: Separate context semantics

o a regular expression r that describes the interaction of the context with o. This r can then be used to refine the approximation of the object semantics. In general, a context creates several objects and it interacts with each of them in a different way. As a consequence, in the definition of the abstract context semantics $\bar{\mathbb{S}}[\![\cdot]\!]$ we use a domain $\dot{\bar{\mathsf{O}}} = [\overline{\mathsf{Addr}} \to \bar{\mathsf{O}}]$, whose elements are maps from abstract objects to their approximations. The definition of $\bar{\mathbb{S}}[\![\texttt{C}]\!]$ is by structural induction on C:

Definition 10.8 (Separate Context Semantics, $\bar{\mathbb{S}}[\![\cdot]\!]$) *The separate context semantics* $\bar{\mathbb{S}}[\![\texttt{C}]\!] \in [\bar{\mathsf{D}} \times \dot{\bar{\mathsf{O}}} \times \dot{\mathsf{R}} \to \bar{\mathsf{D}} \times \dot{\mathsf{R}}]$ *is defined in Figure 10.5.*

Let us describe the semantics in Figure 10.5 informally. It takes three parameters: an abstract state, an approximation of the semantics of the objects and the invocation history. When a class is instantiated, the semantics $\bar{\mathbb{S}}[\![\cdot]\!]$ (abstractly) evaluates the value to pass to the constructor `init` and it obtains an address ϑ for the new object. Then, it uses the object abstraction $\dot{\vartheta}(\vartheta)$ to get the constructor postcondition $\bar{\mathsf{p}}(\texttt{init})$ and it updates the invocation history. In general, the abstract address ϑ identifies a set $\gamma_v(\vartheta)$ of concrete objects. So, the semantics adds an entry to the ϑ history corresponding to the invocation of `init`, with an input $\bar{\mathsf{v}}$ and an abstract state $\bar{\mathsf{d}}$. The result is the new abstract state, obtained considering the store after the execution of the constructor, and the updated invocation history. The sequence, the `skip` and the two assignments do not interact with objects so, in these cases, $\bar{\mathbb{S}}[\![\cdot]\!]$ is very close to the corresponding semantics in Figure 10.2. The definition of $\bar{\mathbb{S}}[\![\cdot]\!]$ for method invocation is similar to the constructor's one: it fetches the (abstract) address corresponding to `o` and the corresponding invariant. Then, it updates the abstract state, using the `m` postcondition, and the invocation history. The definition of the conditional merges the abstract states and the invocation histories originating from the two branches. Eventually, the loop is handled by the least fixpoint operator on the abstract domain $\bar{\mathsf{D}} \times \dot{\mathsf{R}}$. In particular we consider the least fixpoint greater than $(\bar{\mathsf{d}}, \lambda\vartheta.\ \varepsilon)$ as we need to compute an invocation history that is valid for all the iterations of the loop body. The history for the whole `while` command is the concatenation of the input history with the body one, repeated an unspecified number of times. As usual, the convergence of the analysis can be forced through the use of the widening operator

$$\lambda(\bar{\mathsf{d}}_1, \dot{r}_1).(\bar{\mathsf{d}}_2, \dot{r}_2).\ (\bar{\mathsf{d}}_1 \bar{\nabla} \bar{\mathsf{d}}_2, \dot{r}_1 \dot{\nabla}_r \dot{r}_2).$$

We conclude this section with two soundness lemmata. The first one states that for each initial value and object approximation, all the history traces computed by $\bar{\mathbb{S}}[\![\cdot]\!]$ are of the form of $\langle\{\texttt{init}\}, \bar{\mathsf{v}}, \bar{\mathsf{s}}\rangle \cdot r$, for some $\bar{\mathsf{v}} \in \overline{\mathsf{Val}}$, $\bar{\mathsf{s}} \in \bar{\mathsf{D}}$ and regular expression r. Intuitively, it means that the first interaction of the context with an object is the invocation of `init` with some value and store configuration. Formally, the following lemma can be proved by structural induction:

LEMMA 10.5 (SOUNDNESS OF $\bar{\mathbb{O}}[\![\cdot]\!]$) *Let* $\bar{\mathsf{d}}_0 \in \bar{\mathsf{D}}$, $\dot{\vartheta} \in \dot{\mathsf{O}}$ *and*

$$\bar{\mathbb{S}}[\![\mathsf{C}]\!](\bar{\mathsf{d}}_0, \dot{\vartheta}, \lambda\vartheta.\varepsilon) = (\bar{\mathsf{d}}, \dot{r}).$$

Then for all the abstract objects ϑ *such that* $\dot{r}(\vartheta) \neq \varepsilon$*:*

1. $\exists \bar{\mathsf{v}} \in \overline{\mathsf{Val}}.\ \exists \bar{\mathsf{s}} \in \bar{\mathsf{D}}.\ \exists r \in \mathsf{R}.\ \dot{r}(\vartheta)\langle\{\texttt{init}\}, \bar{\mathsf{v}}, \bar{\mathsf{s}}\rangle \cdot r$;

2. $\bar{\mathbb{O}}[\![\vartheta]\!](\dot{r}(\vartheta), \dot{\bot}_o) \bar{\sqsubseteq}_o \bar{\mathbb{C}}[\![\mathtt{A}]\!]$.

The next lemma shows that the history traces computed by $\bar{\mathbb{S}}[\![\cdot]\!]$ are an over-approximation of the history traces computed by $\bar{\mathbb{K}}[\![\cdot]\!]$. Thus, the soundness of $\bar{\mathbb{K}}[\![\cdot]\!]$ implies that the history traces are a sound approximation of the context. The proof of the lemma uses a slight refinement of the semantics in Figure 10.2 in order to deal with the history of (abstract) method invocations.

LEMMA 10.6 (SOUNDNESS OF THE HISTORY TRACES) *Let* $\bar{\mathbb{K}}[\![\mathtt{C}[\mathtt{o}]]\!](\bar{\bot}) = \bar{\mathtt{d}}$, $\alpha_v(\{\mathtt{o}\}) = \vartheta$ *and*

$$t = \langle \mathtt{init}, \bar{\mathtt{v}}, \bar{\mathtt{s}} \rangle \cdot \langle \mathtt{m}_1, \bar{\mathtt{v}}_1, \bar{\mathtt{s}}_1 \rangle \ldots \langle \mathtt{m}_n, \bar{\mathtt{v}}_n, \bar{\mathtt{s}}_n \rangle$$

a sequence of method invocations of ϑ *when the rules of Figure 10.2 are used to derive* $\bar{\mathtt{d}}$. *Then*

$$(\bar{\mathtt{d}}', \dot{r}') = \bar{\mathbb{S}}[\![\mathtt{C}[\mathtt{o}]]\!](\bar{\bot}, \lambda\vartheta.\bar{\mathbb{C}}[\![\mathtt{A}]\!], \lambda\vartheta.\varepsilon)$$

are such that $\bar{\mathtt{d}} \bar{\sqsubseteq} \bar{\mathtt{d}}'$ *and* $\mathcal{L}(t) \subseteq \mathcal{L}(\dot{r}'(\vartheta))$.

10.4.5 Putting It All Together

In this section we show how to combine the two abstract semantic functions $\bar{\mathbb{O}}[\![\cdot]\!]$ and $\bar{\mathbb{S}}[\![\cdot]\!]$ in order to obtain a separate compositional analysis of $\mathtt{C}[\mathtt{o}]$. The functions $\bar{\mathbb{O}}[\![\cdot]\!]$ and $\bar{\mathbb{S}}[\![\cdot]\!]$ are mutually related. The first one takes as input an approximation of the context and it returns an approximation of the object semantics. The second one takes as input an approximation of the objects. It returns an abstract state and, for each abstract object ϑ, an approximation of the context that interacts with ϑ. Then it is natural to handle this mutual dependence with a fixpoint operator.

First, we need to formally define the function $\Theta \in [\dot{\mathsf{R}} \to \dot{\mathsf{O}}]$, that maps an interaction history $\dot{r}$, to a function $\dot{\vartheta}$ from abstract objects to their approximation. Second, we consider the set of the abstract objects that interact with the context, i.e. the abstract addresses whom interaction history is non-empty: $I = \{\vartheta \mid \dot{r}(\vartheta) \neq \varepsilon\}$. Third, we define a function that maps elements of I to their abstract semantics and the others to the class invariant $\bar{\mathbb{C}}[\![\mathtt{A}]\!]$:

$$\dot{\vartheta}_{\dot{r}} = \lambda\vartheta. \begin{cases} \bar{\mathbb{O}}[\![\vartheta]\!](\dot{r}(\vartheta), \bar{\bot}_o) & \text{if } \vartheta \in I \\ \bar{\mathbb{C}}[\![\mathtt{A}]\!] & \textit{otherwise.} \end{cases} \tag{10.1}$$

Finally, we require that the more precise the abstract object, the more precise its abstract semantics. Therefore we perform the downward closure of $\dot{\vartheta}_{\dot{r}}$, to make it monotonic:

Definition 10.9 (Object Abstractions in a Context) *Let* $\dot{r} \in \dot{\mathsf{R}}$ *and let* $\dot{\vartheta}_{\dot{r}}$ *be as* (10.1)*. Then the object abstractions function in a context* $\dot{r}$*,* $\Theta \in [\dot{\mathsf{R}} \to \dot{\bar{\mathsf{O}}}]$*, is defined as follows:*

$$\Theta(\dot{r}) = \lambda\vartheta.\ \bar{\sqcap}_o\{\dot{\vartheta}_{\dot{r}}(\vartheta') \mid \vartheta' \in \overline{\mathsf{Addr}} \text{ and } \vartheta\ \bar{\sqsubseteq}_v \vartheta'\}.$$

The function Θ is well-defined as $\overline{\mathsf{Addr}}$ is a sublattice of $\overline{\mathsf{Val}}$. Moreover, the monotonicity of $\Theta(\dot{r})$ is a direct consequence of the definition.

Using the above definition and putting $\Gamma(\dot{\vartheta}) = \bar{\mathbb{S}}[\![\mathsf{C}]\!](\bar{\bot}, \dot{\vartheta}, \lambda\vartheta.\varepsilon)$, it is now possible to formally state the interdependence between the context and the objects semantics as follows:

$$\begin{aligned} \dot{\vartheta} &= \Theta(\dot{r}) \\ (\bar{\mathsf{d}}, \dot{r}) &= \Gamma(\dot{\vartheta}). \end{aligned} \tag{10.2}$$

A solution to the recursive equation (10.2) can be found with the standard iterative techniques. Nevertheless, our goal is to parallelize the iterative computation of Θ and Γ, in order to speed up the whole analysis. Therefore, we start the iterations by considering a worst-case approximation for $\dot{r}$ and $\dot{\vartheta}$: $\dot{r}_0 = \lambda\vartheta.r_\top$ and $\dot{\vartheta}_0 = \lambda\vartheta.\bar{\top}_o$. In other words, we assume an unknown context when computing the abstract object semantics and an unknown semantics when analyzing the context. Then we obtain $\dot{\vartheta}_1 = \Theta(\dot{r}_0)$ and $(\bar{\mathsf{d}}_1, \dot{r}_1) = \Gamma(\dot{\vartheta}_0)$.

As we consider the worst-case approximation for the objects semantics, the abstract state $\bar{\mathsf{d}}_1$ is an upper approximation of $\bar{\mathbb{K}}[\![\mathsf{C}]\!](\bar{\bot})$ [35]. Furthermore, it is easy to see that $\dot{r}_1 \dot{\bar{\sqsubseteq}}_r \dot{r}_0$ and $\dot{\vartheta}_1 \dot{\bar{\sqsubseteq}}_o \dot{\vartheta}_0$. Roughly speaking, this means that after one iteration we have a better approximation of the context and the object semantics. As a consequence, if we compute $\dot{\vartheta}_2 = \Theta(\dot{r}_1)$ and $(\bar{\mathsf{d}}_2, \dot{r}_2) = \Gamma(\dot{\vartheta}_1)$, we obtain a better approximation for the abstract state, the semantics of the objects and that of the context. This process can be iterated, so that at step $i+1$ we have:

$$\begin{aligned} \dot{\vartheta}_{i+1} &= \Theta(\dot{r}_i) \\ (\bar{\mathsf{d}}_{i+1}, \dot{r}_{i+1}) &= \Gamma(\dot{\vartheta}_i). \end{aligned} \tag{10.3}$$

The next theorem synthesizes what has been said so far. It states that the iterations of (10.3) form a decreasing chain and that at each iteration step $\bar{\mathsf{d}}_{i+1}$ is an sound approximation of the monolithic abstract semantics. Hence, of the concrete semantics.

Theorem 10.2 (Soundness) *Let* C *be a context. Then* $\forall i \geq 0$*.*

1. $\bar{\mathsf{d}}_{i+1} \bar{\sqsubseteq} \bar{\mathsf{d}}_i$*,* $\dot{r}_{i+1} \dot{\bar{\sqsubseteq}}_r \dot{r}_i$ *and* $\dot{\vartheta}_{i+1} \dot{\bar{\sqsubseteq}}_o \dot{\vartheta}_i$*.*

2. $\bar{\mathbb{K}}[\![\mathtt{C}]\!](\bar{\bot}) \bar{\sqsubseteq} \bar{\mathtt{d}}_i$.

Proof. (Sketch) The theorem follows from the fact that (abstract) semantic functions are monotonic and that the dual of (10.3) is an instance of the asynchronous iteration schema (cf. Definition 2.7), so that it produces a chain of increasing elements and because of Theorem 2.9 such a chain converges to the least fixpoint at a rank $\rho \in \mathbb{O}$. Therefore by duality, the chain originated by (10.3) forms a decreasing chain that converges to the greatest fixpoint, and in particular $\bar{\mathtt{d}}_\rho$ is above the least fixpoint $\bar{\mathbb{K}}[\![\mathtt{C}]\!]$.

q.e.d.

Roughly speaking the first point of the theorem states that the more the iterations the more precise the result of the analysis. On the other hand, the second point states that the abstract states are all above the result of the monolithic abstract semantics. Nevertheless, in general the abstract domain may not satisfy the Descending Chain Condition, so that a narrowing operator must be used [29]. For instance, a possible narrowing is to stop the iterations at a step $\underline{i}$. The resulting abstract state $\bar{\mathtt{d}}_{\underline{i}}$ is then a sound approximation of the concrete semantics.

An analysis based on (10.3) has several advantages. First, it is possible to use the asynchronous iterations with memory [24] in order to parallelize the analysis of the context and the objects. Intuitively, this is a consequence of the fact that at each iteration, the result of Θ and Γ depends just on the result of the previous iteration. Furthermore, Θ computes the abstract semantics for several, independent, abstract objects (cf. (10.1)). Therefore, even the effective implementation of Θ may take advantage of a further parallelization. Finally the fact that each iteration is a sound approximation allows a fine tuning of the trade-off precision/cost. In particular, we can stop the iterations as soon as the desired degree of precision is reached.

Example 10.1 As an example, we can consider the context and the class `A` in Figure 10.6, where `Prop` is the property:

$$(\mathtt{o}_1.a + \mathtt{o}_1.b) - (\mathtt{o}_2.a + \mathtt{o}_2.b) + (\mathtt{o}_1.y + \mathtt{o}_2.y) \geq 0.$$

We are interested in proving that the assert condition is never violated. In order to do it, we instantiate the abstract domain $\bar{\mathsf{D}}$ with Polyhedra [37], and we consider the two abstract objects ϑ_1 and ϑ_2 corresponding respectively to $\mathtt{o}_1$ and $\mathtt{o}_2$. According to the iteration schema (10.3), the first step approximates the objects semantics with the class invariant: $\Theta(\dot{r}_0) = \lambda\vartheta.\langle\bar{\mathtt{i}}, \lambda\mathtt{m}.\bar{\mathtt{p}}(\mathtt{m})\rangle$.

$\mathtt{o_1} = \mathtt{new\ A}(5, 10);$
$\mathtt{o_2} = \mathtt{new\ A}(3, 10);$
$\mathtt{while\ \ldots\ do}$
  $\mathtt{if}\ \mathtt{o_1.get\_y}() + \mathtt{o_2.get\_y}() \geq 0\ \mathtt{then}$
    $\mathtt{o_1.addA}(5); \mathtt{o_1.addB}(3);$
  $\mathtt{else}$
    $\mathtt{o_2.addA}(7); \mathtt{o_2.addA}(1);$
$\{\ \mathtt{assert}(\mathtt{Prop})\ \}$

(a) The context

$\mathtt{F} : \{\mathtt{a}, \mathtt{b}, \mathtt{y}\}$

$\mathtt{init}(\mathtt{a_0}, \mathtt{c_0}) : \mathtt{a} = \mathtt{a_0}; \mathtt{b} = \mathtt{c_0} - \mathtt{a_0}; \mathtt{y} = 0$

$\mathtt{addA}(\mathtt{x}) : \mathtt{a} = \mathtt{a} + \mathtt{x}; \mathtt{b} = \mathtt{b} - \mathtt{x}; \mathtt{y} = \mathtt{y} + 1$
$\mathtt{addB}(\mathtt{x}) : \mathtt{a} = \mathtt{a} - \mathtt{x}; \mathtt{b} = \mathtt{b} + \mathtt{x}; \mathtt{y} = \mathtt{y} - 1$
$\mathtt{get\_y}() : \mathtt{return\ y}$

(b) The class A

Figure 10.6: Example of a context and a class

The object fields invariant is $\bar{\mathsf{i}} = \{\mathtt{a} + \mathtt{b} = \mathtt{c_0}\}$ and the method postconditions are:

$$\bar{\mathsf{p}} = \begin{cases} \mathtt{init} \mapsto \langle \bar{\bot}_v, \bar{\mathsf{i}} \cup \{\mathtt{y} = 0\}\rangle \\ \mathtt{addA} \mapsto \langle \bar{\bot}_v, \bar{\mathsf{i}} \cup \{\mathtt{y} = \mathtt{y} + 1\}\rangle \\ \mathtt{addB} \mapsto \langle \bar{\bot}_v, \bar{\mathsf{i}} \cup \{\mathtt{y} = \mathtt{y} - 1\}\rangle \\ \mathtt{get\_y} \mapsto \langle \bar{\top}_v, \bar{\mathsf{i}}\rangle. \end{cases}$$

On the other hand, as far as the context analysis is concerned, we have $(\emptyset, \dot{r}_1) = \Gamma(\dot{\vartheta}_0)$, where $\dot{r}_1$ is the following interaction history [1]:

$$\dot{r}_1 = \begin{cases} \vartheta_1 \mapsto \langle\{\mathtt{init}\}, (5, 10)\rangle \cdot (\langle\{\mathtt{get\_y}\}, \emptyset\rangle \cdot \langle\{\mathtt{addA}\}, 5\rangle \cdot \langle\{\mathtt{addB}\}, 3\rangle)^* \\ \vartheta_2 \mapsto \langle\{\mathtt{init}\}, (7, 10)\rangle \cdot (\langle\{\mathtt{get\_y}\}, \emptyset\rangle \cdot \langle\{\mathtt{addA}\}, 7\rangle \cdot \langle\{\mathtt{addA}\}, 1\rangle)^* \end{cases}.$$

The result of the next iteration, $\Gamma(\dot{\vartheta}_1)$, is still too imprecise for verifying the assertion, as the object fields invariant $\bar{\mathsf{i}}$ implies that $(\mathtt{o_1}.a + \mathtt{o_1}.b) - (\mathtt{o_2}.a + \mathtt{o_2}.b) = 0$, but nothing can be said about $\mathtt{o_1}.y + \mathtt{o_2}.y$. Nevertheless,

[1] We simplify the structure of the interaction history by omitting the abstract state. Nevertheless, in the example this is not problematic, as the objects do not expose the internal state.

the analysis of the object semantics under the context $\dot{r}_1$ results in a more precise approximation of the objects semantics. In particular, we obtain for the first object the field invariant $\bar{\mathsf{i}}_1 = \bar{\mathsf{i}} \cup \{\mathtt{0} \leq \mathtt{y} \leq \mathtt{1}\}$ and for the latter $\bar{\mathsf{i}}_2 = \bar{\mathsf{i}} \cup \{\mathtt{y} \geq \mathtt{0}\}$. As a consequence, a further iteration is enough to infer that the condition `Prop` is verified. From Theorem 10.2 it follows that the result is sound, even if it is not the most precise one. In fact it is easy to see that a further iteration gives a more precise result, proving that the `else` branch in the conditional is never taken. Therefore that $\mathtt{o}_2.\mathtt{y}$ is identically equal to zero.

10.5 Discussion

In this chapter we introduced a separate compositional analysis and we proved it correct for a small yet realistic object-oriented language. In particular we presented an iteration schema for the computation of the abstract semantics that approximates it from above. The central idea for the parallelization is the use of a domain of regular expressions to encode the interactions between the context and the objects.

In future work we plan to study the practical effectiveness of the presented technique, for example with regard to memory consumption. Moreover, it would be interesting to study how many iterations are needed in order to reach an acceptable degree of precisions. As far as the theoretical point of view is concerned, a straightforward extension of this work is a direct handling of inheritance. Nevertheless, in our opinion the combination of the present work with modular techniques for the handling of inheritance of Chapter 8 presents some more challenges that must be explored.

Chapter 11

Conclusions

Finem respice. [1]

Chilon (560 BCE)

We presented a framework for the modular analysis of object-oriented languages. We defined a liberal and generic trace semantics for class-bases object-oriented languages and we proved it sound and complete w.r.t. a trace semantics for object-oriented programs. We derived systematically the equations characterizing class invariants as an abstraction of the class concrete semantics. We dealt with the three main features of object-oriented languages:

- inheritance, by considering the analysis of subclasses without accessing the parent's class source;
- polymorphism, by studying an effective notion of behavioral subtyping;
- encapsulation, by abstracting the interactions between an object and its context using regular expressions.

The framework is very flexible and in particular we can distinguish three orthogonal axes for the analysis:

- abstract domain: a class can be analyzed using either a generic abstract domain (Chapters 6 and 5) or a symbolic relational domain (Chapter 7) to obtain a more efficient analysis;

[1] (Latin) Have regard to the end.

- inheritance: a subclass can be analyzed either directly, by expanding the subclass relation, or indirectly, by using the parent's invariant (Chapters 8 and 9);
- context: a class can be analyzed either aside from the instantiation context, so to obtain a result valid for all the contexts, or using an approximation of the context itself (Chapter 10).

The future work will include the implementation and the study of the practical aspects of our results. In particular the interest will be the exploration of the orthogonal axes of the analysis. Furthermore, we plan to extend the results in order to cope with aspect-oriented languages [38], with concurrency and with temporal properties of objects.

Bibliography

[1] M. Abadi and L. Cardelli. An imperative object calculus. *Theory and Practice of Object Systems*, 1(3):151–166, 1995.

[2] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[3] A. Aggarwal and K. H. Randall. Related field analysis. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, volume 36(5) of *SIGPLAN Notices*, pages 214–220. ACM Press, June 2001.

[4] P. America. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '87)*, volume 276 of *Lectures Notes in Computer Science*, pages 234–242. Springer-Verlag, June 1987.

[5] Apple Inc. The Objective-C programming language. Available online at `http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf`.

[6] K. Arnout and B. Meyer. Finding implicit contracts in .NET components. In *First International Symposium on Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *Lectures Notes in Computer Science*, pages 285–318. Springer-Verlag, October 2002.

[7] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In *Proceedings of the 10th Static Analysis Symposium 2003 (SAS '03)*, volume 2694 of *Lectures Notes in Computer Science*, pages 337–354. Springer-Verlag, June 2003.

[8] B. Blanchet. Escape analysis for object oriented languages. Application to Java. In *14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 20–34. ACM Press, November 1999.

[9] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207. ACM Press, June 2003.

[10] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common LISP object system specification. *SIGPLAN Notices*, 23(Special Issue), September 1988.

[11] E. Börger. The origins and the development of the ASM method for high level system design and analysis. *Journal of Universal Computer Science*, 8, August 2002.

[12] Borland Inc. *Turbo Pascal 5.5 Object Oriented Programming Guide*. Borland Inc., 1989. Available online at http://community.borland.com/ article/ images/20803/TP_55_OOP_Guide.pdf.

[13] G. Bracha and W. R. Cook. Mixin-based inheritance. In *Proceedings of the 5th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '90)*, volume 25(10) of *SIGPLAN Notices*, pages 303–311, October 1990.

[14] M. Campione, K. Walrath, and A. Huml. *The Java Tutorial: A Short Course on the Basics*. Addison-Wesley, third edition, 2000.

[15] G. Cantor. Contributions to the founding of the theory of transfinite numbers (1 and 2). *Mathematische Annalen*, 1895 and 1897. Original in German. French traduction by Éditions Jacques Gabay, Sceaux.

[16] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67, Berlin, 1984. Springer-Verlag. Full version in *Information and Computation*, 76(2/3):138–164, 1988.

[17] L. Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 266–277. ACM Press, 1997.

[18] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '99)*, pages 133–146. ACM Press, 1999.

[19] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[20] R. Clarisó and J. Coradella. The octahedron abstract domain. In *Proceedings of the 11th Static Analysis Symposium (SAS'04)*, Lectures Notes in Computer Science, August 2004.

[21] M. Comini, G. Levi, and M. C. Meo. A theory of observables for logic programs. *Information and Computation*, 169(1):23–80, August 2001.

[22] W. R. Cook, W. Hill, and P. S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'90)*. ACM Press, January 1990.

[23] W. R. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, November 1994.

[24] P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Res. rep. R.R. 88, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France, September 1977.

[25] P. Cousot. Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes. Thèse d'Etat ès sciences mathématiques, Université scientifique et médicale de Grenoble, March 1978.

[26] P. Cousot. Types as abstract interpretations, invited paper. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL '97)*, pages 316–331. ACM Press, January 1997.

[27] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[28] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Journal of Theoretical Computer Science (TCS)*, 277(1–2):47–103, April 2002.

[29] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM Press, January 1977.

[30] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Math.*, 82(1):43–57, 1979.

[31] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '79)*, pages 269–282. ACM Press, 1979.

[32] P. Cousot and R. Cousot. Relational abstract interpretation of higher-order functional programs. JTASPEFL'91, Bordeaux. *BIGRE*, 74:33–36, October 1991.

[33] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, July 1992.

[34] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[35] P. Cousot and R. Cousot. Modular static program analysis, invited paper. In *Proceedings of the Eleventh International Conference on Compiler Construction (CC 2002)*, volume 2304 of *Lectures Notes in Computer Science*, pages 159–178. Springer-Verlag, April 2002.

[36] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, pages 178–190. ACM Press, New York, NY, January 2002.

[37] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '78)*, pages 84–97. ACM Press, 1978.

[38] D. Crawford, editor. *Aspect Oriented Programming*, volume 44(10) of *Communications of the ACM (CACM)*, New York, October 2001. ACM Press.

[39] O. Dahl and K. Nygaard. SIMULA – an ALGOL-based simulation language. *Communications of the ACM (CACM)*, 9(9):671–678, September 1966.

[40] D. Detlefs. Automatic inference of reference-count invariant. In *Proceedings of the 2nd workshop on Semantics, Program Analysis, and Computing Environments For Memory Management*, January 2004. Available at http://www.diku.dk/topps/space2004/space_final/detlefs.pdf.

[41] D. Distefano, J.P. Katoen, and A. Rensik. On a temporal logic for object-based systems. In *4th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000)*, volume 177 of *IFIP Conference Proceedings*, pages 285–304, Stanford, CA, U.S.A., September 2000. Kluwer Academic Publishers.

[42] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '96)*, volume 31(10) of *SIGPLAN Notices*, pages 306–323. ACM Press, October 1996.

[43] B. Eckel. *Thinking in C++, 2nd Edition*, volume 1. Prentice Hall, 2000.

[44] M. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, 2002.

[45] J. Feret. Abstract interpretation of mobile systems. *Journal of Logic and Algebraic Programming, special issue on π-calculus*, 2004.

[46] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe (FME 2001)*, volume 2021 of *Lectures Notes in Computer Science*, pages 500–517. Springer-Verlag, March 2001.

[47] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, pages 234–245. ACM Press, June 2002.

[48] Free Software Foundation. *Nana: improved support for assertions and logging.* GNU, 2003. http://www.gnu.org/manual/nana-1.14/.

[49] Gartner, Inc. .NET vs. Java: Competition or cooperation? Slides are available online at http://www.gartnervoice.com/homepageApril2003/enetvJavaDW.pdf.

[50] S. Genaim and M. Codish. Incremental refinement of semantic based program analysis for logic programs. In *Proceedings of the 22nd Australasian Computer Science Conference*. Springer-Verlag, January 1999.

[51] S. Ghemawat, K. H. Randall, and Scales D. J. Field analysis: getting useful and low-cost interprocedural information. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, volume 35(5) of *ACM SIGPLAN Notices*, pages 334–344. ACM Press, June 2000.

[52] R. Giacobazzi and F. Ranzato. Refining and compressing abstract domains. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP'97)*, volume 1256 of *Lectures Notes in Computer Science*, pages 771–781. Springer-Verlag, 1997.

[53] R. Giacobazzi and F. Ranzato. The reduced relative power operation on abstract domains. *Journal of Theoretical Computer Science (TCS)*, 216(1-2):159–211, March 1999.

[54] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.

[55] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[56] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification - 2nd Edition.* Sun Microsystems, 2001.

[57] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91)*, volume 464 of *Lectures Notes in Computer Science*, pages 169–192. Springer-Verlag, April 1991.

[58] J. V. Guttag, S. J. Horning, J. J. with Garland, K. D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[59] M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Proceedings of the 5th Static Analysis Symposium (SAS '98)*, volume 1503 of *Lectures Notes in Computer Science*, pages 200–215. Springer-Verlag, 1998.

[60] M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's vector class. In *Object-Oriented Technology: ECOOP'99 Workshop Reader*, volume 1743 of *Lecture Notes in Computer Science*, pages 109–110. Springer-Verlag, June 1999.

[61] K. Huizing and R. Kuiper. Verification of object oriented programs using class invariants. In *Fundamental Approaches to Software Engineering, Third International Conference (FASE 2000)*, volume 1783 of *Lectures Notes in Computer Science*, pages 208–221. Springer-Verlag, April 2000.

[62] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 132–146. ACM Press, November 1999.

[63] B. Jacobs, J. van den Berg, H. Huismann, M. van Berkum, U. Hensel, and Tews H. Reasoning about Java classes (preliminary report). In *13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98)*, volume 33(10) of *SIGPLAN Notices*. ACM Press, October 1998.

[64] S. N. Kamin and U. S. Reddy. Two semantic models of object-oriented languages. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. MIT Press, 1994.

[65] O. Kupferman and M. Y. Vardi. Modular model checking. In *Compositionality: The Significant Difference, International Symposium (COMPOS'97). Revised Lectures*, volume 1536 of *Lectures Notes in Computer Science*, pages 381–401. Springer-Verlag, September 1997.

[66] J.L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.

[67] G. T. Leavens, A. L. Baker, and C. Ruby. *Preliminary Design of JML: A Behavioral Interface Specification Language for Java*, November 2003. `ftp://ftp.cs.iastate.edu/pub/leavens/JML/prelimdesign.pdf`.

[68] G. T. Leavens and K. K. Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.

[69] X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, May 2000.

[70] X. Leroy. *The Objective Caml system release 3.07*, 2004. Available at http://caml.inria.fr/ocaml/htmlman/index.html.

[71] J. Lewis and W. Loftus. *Java Software Solutions, Second Edition Update*. Addison Wesley Longman Inc., 2001.

[72] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., 2nd edition, April 1999.

[73] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, November 1994.

[74] F. Logozzo. Class-level modular analysis for object oriented languages. In *Proceedings of the 10th Static Analysis Symposium 2003 (SAS '03)*, volume 2694 of *Lectures Notes in Computer Science*, pages 37–54. Springer-Verlag, June 2003.

[75] F. Logozzo. An approach to behavioral subtyping based on static analysis. In *Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004)*, Electronic Notes in Theoretical Computer Science. Elsevier Science, April 2004.

[76] F. Logozzo. Approximating module semantics with constraints. In *Proceedings of the 19th ACM SIGAPP Symposium on Applied Computing (SAC 2004)*, pages 1490–1495. ACM Press, March 2004.

[77] F. Logozzo. Automatic inference of class invariants. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, volume 2937 of *Lectures Notes in Computer Science*, pages 211–222. Springer-Verlag, January 2004.

[78] F. Logozzo. Separate compositional analysis of class-based object-oriented languages. In *Proceedings of the 10th International Conference on Algebraic Methodology And Software Technology (AMAST'2004)*, volume 3116 of *Lectures Notes in Computer Science*, pages 332–346. Springer-Verlag, July 2004.

[79] L. Mauborgne. Abstract interpretation using typed decision graphs. *Science of Computer Programming (SCP)*, 31(1):91–112, May 1998.

[80] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[81] B. Meyer. *Object-Oriented Software Construction (2nd Edition)*. Professional Technical Reference. Prentice Hall, 1997.

[82] Microsoft Inc. The .net framework. http://msdn.microsoft.com/netframework/.

[83] Microsoft Inc. Visual C++ .net. http://msdn.microsoft.com/visualc/.

[84] Microsoft Inc. *Microsoft C# Language Specifications*. Microsoft Press, 2001.

[85] Microsoft Inc. *The Component Object Model Specification*. Microsoft, 2003. Available online at http://www.microsoft.com/com/tech/com.asp.

[86] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001. http://www.di.ens.fr/~mine/publi/article-mine-ast01.pdf.

[87] NetBeans.org and Sun Mycrosystem, Inc. Netbeans IDE, 2004. http://www.netbeans.org/.

[88] E. Noether. Idealtheorie in ringbereichen. *Mathematical Annals*, 83:24–66, 1921.

[89] T. Owen and D. Watson. Reducing the cost of object boxing. In *Proceedings of the 13th International Conference on Compiler Construction (CC 2004)*, volume 2985 of *Lectures Notes in Computer Science*, pages 202–216. Springer-Verlag, March 2004.

[90] J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, Chichester, 1994.

[91] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, bo 2000.

[92] I. Pollet, B. Le Charlier, and A. Cortesi. Distinctness and sharing domains for static analysis of Java programs. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP '01)*, volume 2072 of *Lectures Notes in Computer Science*, pages 77–98. Springer-Verlag, 2001.

[93] C. Probst. Modular control flow analysis for libraries. In *Proceedings of the 9th Static Analysys Symposium (SAS '02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 165–179. Springer-Verlag, 2002.

[94] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 245–255. ACM Press, January 2004.

[95] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, volume 37(5) of *ACM SIGPLAN Notices*, pages 83–94. ACM Press, June 2002.

[96] A. Rountev, A. Milanova, and B.G. Ryder. Fragment class analysis for testing of polymorphism in Java software. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 210–220. IEEE, May 2003.

[97] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, Massachusetts, USA, 1st edition, 1999.

[98] D. S. Scott. Domains for denotational semantics. In *Proceedings of 9th International Colloquium on Automata, Languages and Programming*

(ICALP'82), volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer-Verlag, 1982.

[99] F. Spoto and T. P. Jensen. Class analyses as abstract interpretations of trace semantics. *ACM Transactions on Programming Languages and Systems*, 25(5):578–630, September 2003.

[100] R. Stärk, J. Schmid, and E. Börger. *Java and the JavaVirtual Machine: Definition, Verification, Validation*. Springer-Verlag, Berlin, 2001.

[101] B. Stroustrup. *The C++ Programming Language (Special 3rd Edition)*. Addison-Wesley, 2000.

[102] Sun Microsystem, Inc. *javadoc Tool Homepage*, 2004. http://java.sun.com/ j2se/javadoc/.

[103] Sun Microsystems, Inc. JavaCard Technology Homepage, 2004. http://java.sun.com/products/javacard/index.jsp.

[104] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[105] K. Zee and M. Rinard. Write barrier removal by static analysis. In *17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '02)*, volume 37(11) of *SIGPLAN Notices*, pages 191–210. ACM Press, November 2002.